2022 Summer Intern Research Report

The enhancement on Anti-Evasion Techniques in Cuckoo Monitor

Chung-Yen Chiang

Institute of Information Science, Academia Sinica

July 2022 to August 2022

Contents

1 Introduction

Some malware programs have techniques to prevent themselves from being analyzed, so as to make not only static but dynamic analyzer tools become useless.

For example, malware program designer uses some tools, like ConfuserEx, to obfuscate their code. Therefore, when analysts are trying to perform reverse engineering to these malware programs, they have to spend a lot of efforts on deobfuscating the obfuscated code.

As for dynamic analysis, malware programs have other techniques to make themselves invisible. If malware programs are inside virtual machines or sandboxes, they may alter their behavior to be benign or conceal core functions, which is called sandbox/VME evasion. Commonly, malware programs check whether System Information, Registry, Hardware and etc. contain some special values, to confirm that they are inside or outside virtual machines or sandboxes.

If malware programs behave like benign programs when they are inside virtual machines or sandboxes, it is difficult for dynamic analysis tools (Cuckoo Sandbox) to record their behaviors. Therefore, I have to figure out how to prevent these malware programs from evading from being recorded.

I first not only performed reverse engineering to a malware program, but also reviewed related websites, in order to understand what kind of evasion techniques were commonly used. Then, I attempted to delve into how Cuckoo Sandbox worked, and found a way to prevent malware programs from being recorded, which enhanced the analysis ability of Cuckoo Monitor. Finally, I proposed a framework to simplify the redundant procedures that, when finding new evasion techniques, analysts appended these techniques into the source code and compiled.

2 Related Work

2.1 Evasion Techniques Investigation

2.1.1 MITRE ATT&CK

According to T1497 Virtualization/Sandbox Evasion on MITRE ATT&CK website, there are three techniques mentioned, which are T1497.001 System Checks, T1497.002 User Activity Based Checks, T1493.003 Time Based Evasion respectively, and I refer to their descriptions and excerpt them as below.

1. T1497.001 System Checks

   Specific checks will vary based on the target and/or adversary, but may involve behaviors such as Windows Management Instrumentation, PowerShell, System Information Discovery, and Query Registry to obtain system information and search for VME artifacts. Adversaries may search for VME artifacts in memory, processes, file system, hardware, and/or the Registry. Adversaries may use scripting to automate these checks into one script and then have the program exit if it determines the system to be a virtual environment.

2. T1497.002 User Activity Based Checks

   Adversaries may search for user activity on the host based on variables such as the speed/frequency of mouse movements and clicks, browser history, cache, bookmarks, or number of files in common directories such as home or the desktop. Other methods may rely on specific user interaction with the system before the malicious code is activated, such as waiting for a document to close before activating a macro or waiting for a user to double click on an embedded image to activate.

3. T1497.003 Time Based Evasion

   Adversaries may employ various time-based evasions, such as delaying malware functionality upon initial execution using programmatic sleep commands or native system scheduling functionality (ex: Scheduled Task/Job).

Delays may also be based on waiting for specific victim conditions to be met (ex: system time, events, etc.) or employ scheduled Multi-Stage Channels to avoid analysis and scrutiny.

2.1.2 Check Point Research

In addition to MITRE ATT&CK, I also review several evasion techniques mentioned in Check Point Research website and excerpt them as below. Take registry for an example, the principle of all the registry detection methods is the following: there are no such registry keys and values in usual host. However they exist in particular virtual environments. Sometimes usual system may cause false positives when these checks are applied because it has some virtual machines installed and thus some VM artifacts are present in the system. Though in all other aspects such a system is treated clean in comparison with virtual environments.

Table 1: Detection Tables Excerpted From Check Point Research

Detect	Registry path	Details (if any)
[**general**]	HKLM\Software\Classes\Folder\shell\sandbox	
Hyper-V	HKLM\SOFTWARE\Microsoft\Hyper-V	
	HKLM\SOFTWARE\Microsoft\VirtualMachine	
...	...	

2.1.3 Reverse Engineering

GravityRAT is listed in T1497.001 on MITRE ATT&CK website, so I took this malware program as reverse engineering sample, to understand how malware program commonly evaded from being detected.

GravityRAT aims at stealing MAC address, Computer Name, Username, IP address, MS Office files and pdf files. And GravityRAT has 4 variant versions, as GravityRAT G1, G2, G3 and GX respectively. From G2, GravityRAT starts using sandbox evasion attempts.

Therefore, I commenced the analysis of sandbox evasion technique in the final version GX of GravityRAT. I first submitted GravityRAT GX to Cuckoo Sandbox for retrieving the dynamic analysis report, and the report of course did not record any useful behaviors about GravityRAT GX due to its sandbox evasion techniques.

In order to clarify how GravityRAT GX evaded from being detected, I used dnSpy, a .NET reverse engineering tool, to probe in its source code. After performing reverse engineering to GravityRAT GX, I found that there were seven sandbox evasion techniques revealed in the source code.

According to **Figure 1**, GravityRAT GX opened "SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters" registry, which only existed in virtual machines. If this registry could be opened, GravityRAT further queried for its value.

```
RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SOFTWARE\\Microsoft\\Virtual Machine\\Guest\\Parameters");
if (registryKey != null)
{
    virtualMachine = new VirtualMachine(registryKey.GetValue("HostName").ToString(), registryKey.GetValue("VirtualMachineName").ToString());
    result = true;
}
```

Figure 1: The First Evasion Technique

GravityRAT GX not only opened Registry, but performed Windows Management Instrumentation from **Figure 2** to **Figure 6**. According to **Figure 2**, GravityRAT GX queried with "select $*$ from WIN32_BIOS" syntax to obtain BIOS information, and checked if values contained "VMWARE", "Virtual", "XEN" and the other special strings appearing in virtual machine.

```
ManagementObjectCollection.ManagementObjectEnumerator enumerator = new ManagementObjectSearcher("select * from Win32_BIOS").Get().GetEnumerator();
if (!enumerator.MoveNext())
{
    throw new Exception("Unexpected WMI query failure");
}
string text = enumerator.Current["version"].ToString();
enumerator.Current["SerialNumber"].ToString();
string[] array = new string[]
{
    "VMware",
    "Virtual",
    "XEN",
    "Xen",
    "A M I"
};
```

Figure 2: The Second Evasion Technique

According to **Figure 3**, GravityRAT GX queried with "select ∗ from WIN32_ComputerSystem" syntax to obtain Computer System information, and checked if the model column contained "Virtual" or the manufacturer contained "vmware" or not.

```
using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
{
    using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
    {
        foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
        {
            string text = managementBaseObject["Manufacturer"].ToString().ToLower();
            if ((text == "microsoft corporation" && managementBaseObject["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL")) || text.Contains("vmware") ||
            {
                virtualMachine.Host = managementBaseObject["Model"].ToString();
                virtualMachine.MachineName = managementBaseObject["Manufacturer"].ToString();
                return true;
            }
        }
    }
}
```

Figure 3: The Third Evasion Technique

According to **Figure 4**, GravityRAT GX queried with "select ProcessID from Win32_Processor" syntax to obtain ProcessID information, and checked if ProcessorID was null or not.

```
string queryString = "SELECT ProcessorId FROM Win32_Processor";
try
{
    using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = new ManagementObjectSearcher(queryString).Get().GetEnumerator())
    {
        if (enumerator.MoveNext())
        {
            ManagementObject managementObject = (ManagementObject)enumerator.Current;
            if ((string)managementObject["ProcessorId"] == null)
            {
                string host = (string)managementObject["DeviceID"];
                string machineName = (string)managementObject["SystemName"];
                virtualMachine = new VirtualMachine(host, machineName);
                result = true;
                return result;
            }
            result = false;
            return result;
        }
    }
```

Figure 4: The Fourth Evasion Technique

According to **Figure 5**, GravityRAT GX queried with "select ∗ from Win32_Processor" syntax to obtain Processor information, and check if the core number of processor was less than or equal to one or not.

```
using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = new ManagementObjectSearcher("Select * from Win32_Processor").Get().GetEnumerator())
{
    if (enumerator.MoveNext())
    {
        ManagementBaseObject managementBaseObject = enumerator.Current;
        num += int.Parse(managementBaseObject["NumberOfCores"].ToString());
        if (num == 1)
        {
            string host = managementBaseObject["DeviceID"].ToString();
            string machineName = managementBaseObject["SystemName"].ToString();
            virtualMachine = new VirtualMachine(host, machineName);
            result = true;
            return result;
        }
        result = false;
        return result;
    }
}
```

Figure 5: The Fifth Evasion Technique

According to **Figure 6**, GravityRAT GX queried with "select ∗ from MSAcpi_ThermalZoneTemperature" syntax to obtain ThermalZone information, and checked if the environment was able to query CPU Temperature or not.

Figure 6: The Sixth Evasion Technique

Finally, according to **Figure 7**, GravityRAT GX looked up MAC address in the environment, if MAC address contained some special formats, like "00:50:56", "00:0C:29", "00:05:69" and etc., this meant that the malware program was inside virtual machines.

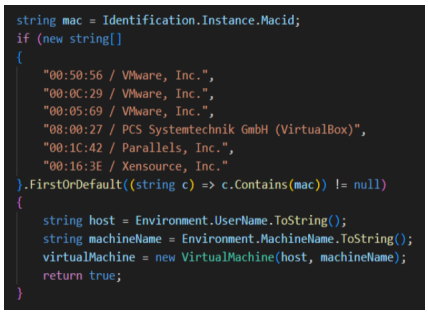

Figure 7: The Seventh Evasion Technique

In summary, if GravityRAT GX was inside virtual machines, it would terminate itself to prevent from being recorded.

2.2 Cuckoo Sandbox

Cuckoo Sandbox is an open-source dynamic analysis tool, which analysts can submit malicious programs to it, waiting for a few minutes, then Cuckoo Sandbox will generate the analysis reports about behaviors of malicious programs.

Before attempting to modify source code of Cuckoo Sandbox to enhance the analysis ability, it is necessary to delve into the work flow of Cuckoo Sandbox. I

excerpt the diagram of Cuckoo Sandbox as **Figure 8** from the official website. Cuckoo Sandbox is divided into two parts, the first is Cuckoo host, and the second is Cuckoo guest.Cuckoo host is responsible for managing multiple virtual machines which are called Cuckoo guest, and these virtual machines are isolated from host because malware programs should be analyzed in a safe environment in order not to infect host environment.

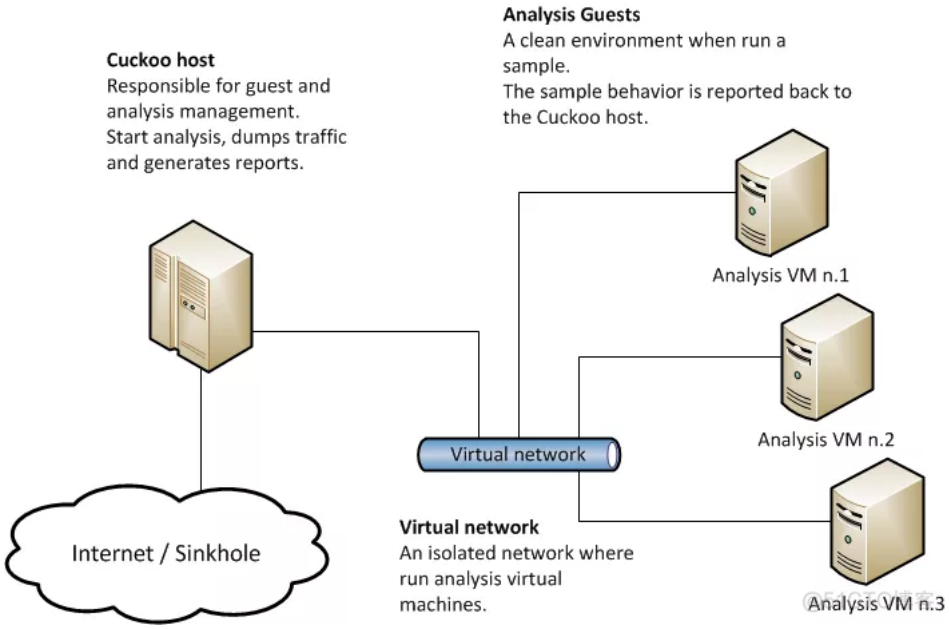

Figure 8: Cuckoo Sandbox Diagram

When analysts submit samples to Cuckoo host, Cuckoo host will first open new virtual machines with the snapshot containing Cuckoo python server-agent scripts, which communicates with Cuckoo host.

After opening new virtual machines, Cuckoo host will submit samples to these virtual machines, and also injects *monitor.dll* into samples. Briefly speaking, *monitor.dll* is used to replace original functions with Cuckoo's trampolined functions, therefore, samples will execute Cuckoo's trampolined functions first, which records behaviors during this time. After executing them, they will redi-

rect samples to original functions.

Besides, I attempted to implement DLL injection to understand how Cuckoo Sandbox actually injected *monitor.dll* into samples. Our attempt was to inject a DLL file (**Figure 9**) into browser, which opened calculator and showed "Injected" when it was successful injected.

```
BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH: {
        WinExec("calc.exe", 1);
        MessageBox(0, TEXT("Injected"), TEXT("Process Attach"), MB_ICONINFORMATION);
        break;
    }
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

Figure 9: Malicious DLL File

The way to implement DLL injection was to first open the process (**Figure 10**) that I wanted to inject into, then I would retrieve a handle *hprocess*. I afterwards took advantage of *hprocess* to allocate the virtual memory space (**Figure 11**) with *hprocess* and its size and DLL path. After I retrieved the virtual memory space *procdlladdr*, I wrote *hprocess* (**Figure 11**) into it, and also got address *loadfunaddr* (**Figure 12**), which was function address of LoadLibraryA, used as a parameter in CreateRemoteThread. Finally, I created a remote thread (**Figure 13**) by using *loadfunaddr* to load *procdlladdr* again to finish DLL injection.

```
char dll_dir[150] = "C:\\Users\\MA\\Desktop\\maldll\\x64\\Debug\\maldll.dll";
DWORD pid = 5388;

int main()
{
    HANDLE hprocess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

    int size = strlen(dll_dir) + 5;
    PVOID procdlladdr = VirtualAllocEx(hprocess, NULL, size, MEM_COMMIT, PAGE_READWRITE);
    if (procdlladdr == NULL) {
        printf("handle %p VirtualAllocEx Failed\n", hprocess);
        return 0;
    }
```

Figure 10: Open Process

```
SIZE_T writenum;
if (!WriteProcessMemory(hprocess, procdlladdr, dll_dir, size, &writenum)) {
    printf("handle %p WriteProcessMemory failed\n", hprocess);
    return 0;
}
```

Figure 11: Write into The Allocated Memory

```
FARPROC loadfuncaddr = GetProcAddress(GetModuleHandle(L"kernel32.dll"), "LoadLibraryA");
if (!loadfuncaddr) {
    printf("handle %p GetProcAddress failed\n", hprocess);
    return 0;
}
```

Figure 12: Get The Address of LoadLibraryA

```
HANDLE hthread = CreateRemoteThread(hprocess, NULL, 0, (LPTHREAD_START_ROUTINE)loadfuncaddr, (LPVOID)procdlladdr, 0, NULL);
if (!hthread) {
    printf("handle %p CreateRemoteThread failed\n", hprocess);
    return 0;
}
printf("handle %p Injection done, WaitForSingleObject return %d\n", hprocess, WaitForSingleObject(hthread, INFINITE));
```

Figure 13: Create a Remote Thread

3 Experiment and Result

3.1 Cuckoo Monitor Modification

Since malware programs query for some special values or artifacts only existing in virtual machines, they have to query them via System APIs, like RegOpenKey, RegQueryValue, ReadFile and etc.

Therefore, I found a tricky way, which was to return fake values after trampolined functions, to prevent malware programs from evading from being detected. For example, malware programs checked whether "SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters" registry could be opened or not, if I returned a fake value while trampolined RegOpenKey was executed, then malware programs were not able to open such registries. Moreover, malware programs further queried for the value of "SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters" registry, if I returned a unexpected value while trampolined RegQueryValue was executed, then malware programs were also not able to query such registries.

To verify the idea was feasible or not, I reviewed a lot of information of how to modify source code of Cuckoo Sandbox. The most significant information I found was an issue on GitHub (**Figure 14**). A people asked about how to actually hook a function with the monitor, and a kind people answered to this question, he gave a significant hint that the hooks were defined in the $*.rst$ files under *sigs* folder, the correct syntax for doing so was documented on Cuckoo Monitor website (**Figure 15**).

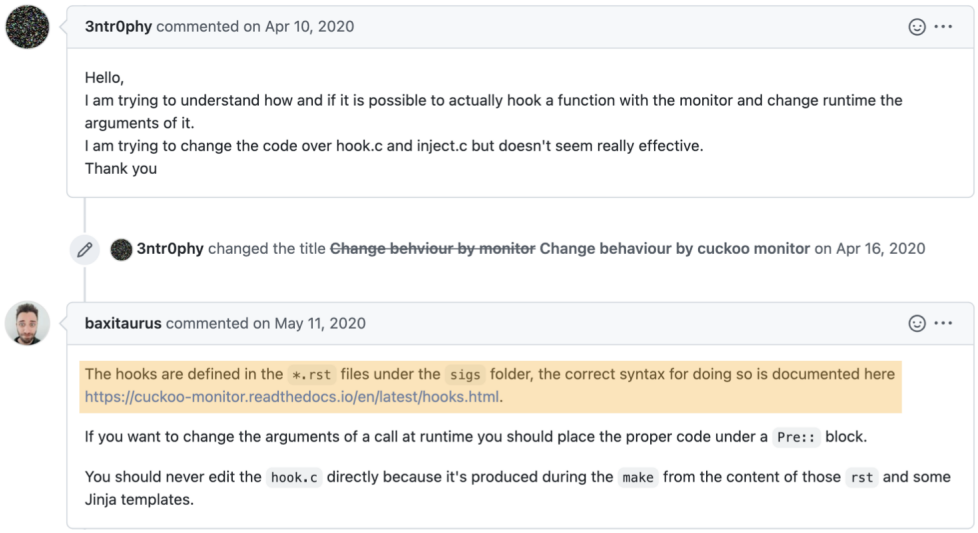

Figure 14: A GitHub Issue about Hooking

Signature Format

The `signature format` is a very basic `reStructuredText`-based way to describe an API signature. This signature is pre-processed by `utils/process.py` to emit C code which is in turn compiled into the Monitor.

Following is an example signature of `system()`:

```
system
======

Signature::

    * Calling convention: WINAPI
    * Category: process
    * Is success: ret == 0
    * Library: msvcrt
    * Return value: int

Parameters::

    ** const char *command
```

Figure 15: Documented on Cuckoo Monitor Website

After understanding that *rst* file was a significant hint, I started to trace source code of Cuckoo Monitor. First, *rst* file was a formatted file, recording details and behaviors of trampolined functions, then it would be generated into a part of variables in *hooks.c*. The most important variable was *sig_hooks*, which was an array of trampolined functions. Cuckoo Monitor would replace original functions with trampolined functions in this array, therefore, this was the reason why both of that kind people and the documentation recommended to modify *rst* file instead of source c files.

However, there were a total of 27 *rst* files (**Figure 16**) should be modified, therefore, I only focused on *Registry.rst* first.

Figure 16: rst Files Listed in sig Folder

Based on the aforementioned information, I attempted to modify the source code of *Registry.rst*. In the format of *rst* files, section of parameters was shown in the beginning of trampolined functions (**Figure 17**), consisting of several required parameters. For example, RegQueryValueExA had 6 parameters, hence, trampolined RegQueryValueExA must have 6 substitute parameters as well. Besides, these trampolined functions must follow the documentations of original functions on Microsoft API website.

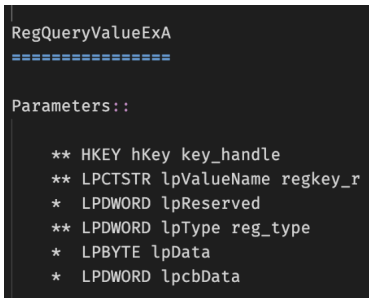

Figure 17: Parameters of Trampolined RegQueryValueExA

In trampolined RegQueryValueExA, lpData is the returned value, which should be obfuscated to make malware programs misjudge. I attempted to add a snippet (**Figure 18**) into RegQueryValueExA, which encrypted the return value with CAESAR Encryption (Offset = 1), to verify whether this attempt worked or not. After modifying this file, it is necessary to execute *make* command in the root directory of Cuckoo Monitor to recompile all files into monitor.dll (**Figure 19**).

```
LPBYTE fake = lpData;
for (int i = 0; *fake != '\0'; i += 2, fake += 2) {
    *(lpData+i) = *(lpData+i) + 1;
}
ret = ERROR_SUCCESS;
```

Figure 18: Snippet Obfuscating the Returned Value

Figure 19: Compiled monitor.dll

According to the hash code in the file of *.cuckoo/monitor/latest*, which was a text file, there existed a folder that its name was identical to the hash code, then replaced *monitor-x86.dll* and *monitor-x64.dll* with compiled dll files. After

finishing these processes, it was time to submit samples with modified dll files.

For example, I wrote a program to query the value of *ProcessNameString*. With original dll files, its returned value was *AMD Ryzen 9 3900X 12-Core Processor* (**Figure 20**). Meanwhile, with modified dll files, its returned value became to be an obfuscated value (**Figure 21**), which made malware program misjudge.


```
{
    "category": "registry",
    "status": 1,
    "stacktrace": [],
    "api": "RegQueryValueExW",
    "return_value": 0,
    "arguments": {
        "key_handle": "0x000000000000003c",
        "value": "AMD Ryzen 9 3900X 12-Core Processor            ",
        "regkey_r": "ProcessorNameString",
        "reg_type": 1,
        "regkey": "HKEY_LOCAL_MACHINE\\HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0\\ProcessorNameString"
    },
    "time": 1659352540.144146,
    "tid": 4176,
    "flags": {
        "reg_type": "REG_SZ"
    }
},
```


Figure 20: Return Value with Original monitor.dll


```
{
    "category": "registry",
    "status": 1,
    "stacktrace": [],
    "api": "RegQueryValueExW",
    "return_value": 0,
    "arguments": {
        "key_handle": "0x000000000000003c",
        "value": "BNE!Sz{fo!:!4:11Y!23.Dpsf!Qspdfttps!!!!!!!!!!!",
        "regkey_r": "ProcessorNameString",
        "reg_type": 1,
        "regkey": "HKEY_LOCAL_MACHINE\\HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0\\ProcessorNameString"
    },
    "time": 1659367898.097271,
    "tid": 1592,
    "flags": {
        "reg_type": "REG_SZ"
    }
},
```


Figure 21: Returned Value with Modified monitor.dll

3.2 Anti-Evasion Application Framework

However, there are not only many evasion techniques varying, but many up-to-date CTI (Cyber Threat Intelligence) Report released all the time. Therefore, as long as new techniques are found by analysts, they have to manually insert these into the corresponding *rst* files to prevent malware programs from evading from being detected. In other words, they spend a lot of time in modifying these up-to-date evasion techniques instead of analyzing malware programs themselves. Briefly speaking, these cumbersome procedures actually obscures analysts main works.

Currently, typical procedures that append a SINGLE new evasion technique are cumbersome as follows.

1. Connect to the connect environment and open *rules.h*.
2. Append a new evasion techniques into *rules.h*
3. Compile them into *monitor.dll* and *detector.exe*
4. Copy them to host and replace the original *monitor.dll* and submit.

- *rules.h* is a header file containing different types of evasion techniques, which *hooks.c* can include this single file to implement unfinished anti-evasion techniques.
- *detector.exe* is an executable file, which pretends to be a malware program to check if anti-evasion techniques according to *rules.h* works or not.

As a consequence, I proposed an anti-evasion application framework (**Figure 22**) to reduce the redundant procedures, which was to connect, compile, copy and replace manually. According to the data path of this framework, it could be more clear to find out where to simplify. Originally, *rules.h* should be updated manually, as well both of *monitor.dll* and *detector.exe* should be compiled manually. However, this framework adopted a client-server architecture to communicate between host and compile environment, so as to reduce the procedures that analysts manually connected.

In this framework, sqlite3 was adopted as a main database to store evasion techniques to further automatically generate *rules.h*. In addition, *server.py* waited for commands from *client.py*, including *compile* that automatically compiled files into *monitor.dll* and *detector.exe* and return them back to client side, *insert* that analysts sorted up-to-date evasion techniques as a json file and type some easy commands on *client.py* to finish inserting.

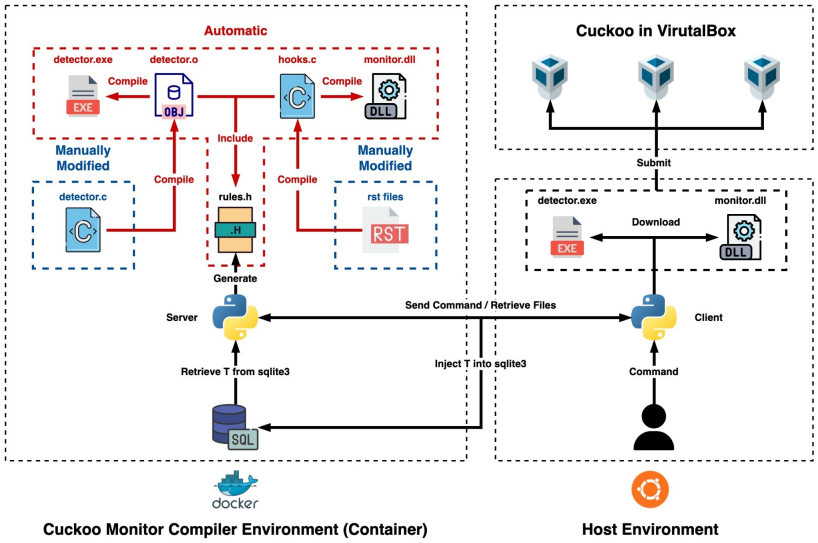

Figure 22: Anti-Evasion Application Framework

Despite the fact that this framework solved most cumbersome procedures, there still existed some manual and one-time procedures that analysts had to do. As I mentioned earlier, there were a total of 27 *rst* files, every APIs in these files related to virtual machines or sandbox evasion should be completed. Against to anti-evasion techniques in *rst* files, *detector.c* pretending to be a malware program to check if these evasion techniques worked or not should be completed as well. As long as these one-time procedures are completed, there is nothing manual to revise anymore. Next, I will introduce the details of this framework as follows.

3.2.1 Evasion Techniques Dataset

I sorted some common evasion techniques of Registry from different websites (Check Point, CTI Reports), and **Figure 23** was a part of sample dataset. To analysts, they can easily sort different types of evasion techniques as a bundle into a json file, then insert this json file into server. Meanwhile, a new corresponding *rules.h* will be automatically generated, like **Figure 24**.

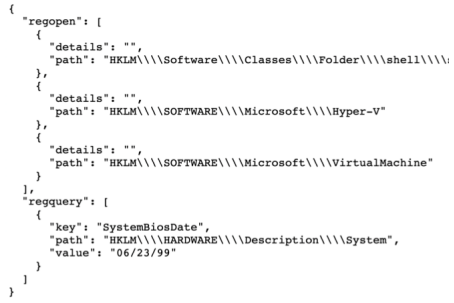

Figure 23: Dataset of Registry Evasion Techniques

Figure 24: Automatically Generated rules.h

3.2.2 detector.c

I finished RegOpenKey and RegQueryValue parts to examine whether anti-evasion techniques worked or not. First, *detector.c* had to include *rules.h*, then assigned two pointers, which *ptr* was the first element of evasion techniques

array and *endptr* was the last element of evasion techniques array (**Figure 25**), to iterate this evasion techniques array.

```
regopen const *ptr = &evasion_regopen[0];
regopen const *endptr = ptr + sizeof(evasion_regopen) / sizeof(evasion_regopen[0]);
```

Figure 25: Pointer Assignment

Because path of registry was formatted, it was necessary to check the prefix and assign its corresponding handle, like **Figure 26**. Then, this program opened (**Figure 27**) and queried (**Figure 28**) for registries.

```
// Check prefix
if (*(lpSubKeyCstr + 2) == 'C' && *(lpSubKeyCstr + 3) == 'U')
{
    hKey = HKEY_CURRENT_USER;
}
else if (*(lpSubKeyCstr + 2) == 'U')
{
    hKey = HKEY_USERS;
    --indexToDel;
}
else if (*(lpSubKeyCstr + 2) == 'L' && *(lpSubKeyCstr + 3) == 'M')
{
    hKey = HKEY_LOCAL_MACHINE;
}
else if (*(lpSubKeyCstr + 2) == 'C' && *(lpSubKeyCstr + 3) == 'R')
{
    hKey = HKEY_CLASSES_ROOT;
}
else if (*(lpSubKeyCstr + 2) == 'C' && *(lpSubKeyCstr + 3) == 'C')
{
    hKey = HKEY_CURRENT_CONFIG;
}
else
{
    hKey = HKEY_LOCAL_MACHINE;
}
```

Figure 26: Prefix Check

```
// And remove prefix
memmove(lpSubKeyMoved, lpSubKeyCstr+indexToDel, strlen(lpSubK

LPCTSTR lpSubKey = TEXT(lpSubKeyMoved);
REGSAM samDesired = KEY_READ|regflag;
HKEY phkResult;

LONG ret;

ret = RegOpenKeyEx(hKey, lpSubKey, 0, samDesired, &phkResult)
printf("Registry Open %s", lpSubKeyCstr);

printf(", ret = %s\n", ret == ERROR_SUCCESS ? "True" : "False

ptr++;
```

Figure 27: Open Registries

```
if (ret == ERROR_SUCCESS)
{
    DWORD lpType = REG_SZ;
    WCHAR lpData[MAX_PATH] = {'\0'};
    DWORD lpcbData = 1024;
    ret = RegQueryValueEx(phkResult, TEXT(lpValueName), 0, &lpType, (LPBYTE)&
    if (ret == ERROR_SUCCESS)
    {
        printf("Registry Query %s, Key = %s, Value = %s\n", lpSubKeyCstr, lpV
    }
    else
    {
        printf("Registry Query %s, Key = %s, Value = %s\n", lpSubKeyCstr, lpV
    }
}
else
{
    printf("Registry Query %s, Key = %s Failed\n", lpSubKeyCstr, lpValueName)
}
```

Figure 28: Query Registries

3.2.3 Registry.rst

The point of anti-evasion techniques in *Registry.rst* was to return a fake value. Hence, I only introduce some important parts of *Registry.rst* because modifying *Registry.rst* was roughly same as modifying *detector.c*. First, according to **Figure 25**, this program had to assign pointers to iterate all elements in evasion techniques array, then both formatted path and given regkey into char[] because some functions of wchar_t were not commonly supported in C99, as

well removed the prefix because the prefix of given regkey (without formatted) and path (with formatted) was different, like **Figure 29**. Also, formatted these char array into lower case (**Figure 30**) because strcmp() is case sensitive.

```c
char path_c[256] = {'\0'};
sprintf(path_c, "%ls", ptr->path);

index = 0;
while (*(path_c+index) != 92)
    ++index;
memmove(path_c, path_c+index+1, strlen(path_c));
```

```c
char regkey_c[256] = {'\0'};
sprintf(regkey_c, "%ls", regkey);

int index = 0;
while (*(regkey_c+index) != 92)
    ++index;
memmove(regkey_c, regkey_c+index+1, strlen(regkey_c));
```

Figure 29: Preprocessing of Parameters

```c
char *ptr_c = path_c;
for (; *ptr_c; ++ptr_c)
    *ptr_c = tolower(*ptr_c);

ptr_c = regkey_c;
for (; *ptr_c; ++ptr_c)
    *ptr_c = tolower(*ptr_c);
```

Figure 30: To Lower Case Processing

If the condition that strcmp and the return equal to ERROR_SUCCESS were simultaneously true, then obfuscated the return to ERROR_NOT_FOUND, so as to make malware programs misjudge, like **Figure 31**. Furthermore, if the condition were true as well in RegQueryValue, then obfuscated the result with CAESAR Encryption, like **Figure 32**.

```
if (strcmp(path_c, regkey_c) == 0 && ret == ERROR_SUCCESS)
{
    ret = ERROR_FILE_NOT_FOUND;
    break;
}
```

Figure 31: Obfuscating during RegOpenKey

```
LPBYTE fake = lpData;
for (int i = 0; *fake != '\0'; i += 2, fake += 2)
{
    *(lpData+i) = *(lpData+i) + i * 10;
}
break;
```

Figure 32: Obfuscating during RegQueryValue

3.2.4 Result

After modifying a part of these necessary one-time files, I experimented on this framework. First, I typed *python3 client.py –dl* to automatically compile files and retrieve *monitor.dll* and *detector.exe* back, like **Figure 33**.

```
cuckoo@L307:~/Desktop/cuckoomon$ python3 client.py -dl
Retrieving files now...
Finished
Replacing now...
monitor-x64.dll
monitor-x86.dll
detector-x64.exe
detector-x86.exe
Finised
```

Figure 33: Commands

Then, I submitted *detector.exe* to Cuckoo Sandbox, and observed its behaviors. As **Figure 34** and **Figure 35** shown below, there were 6 RegOpenKey and 3 RegQueryValue evasion techniques detected with original *monitor.dll*.

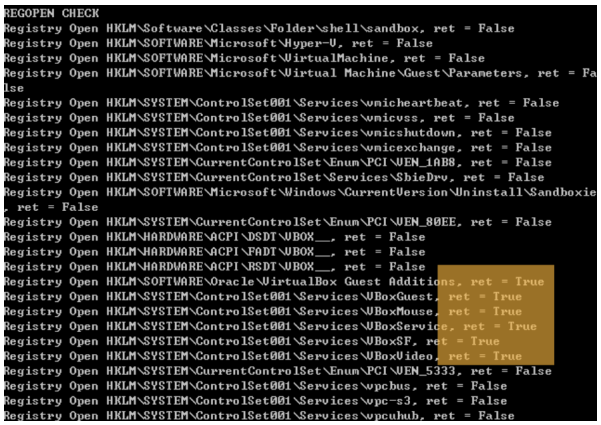

Figure 34: Result of Opening Registries with Original *monitor.dll*

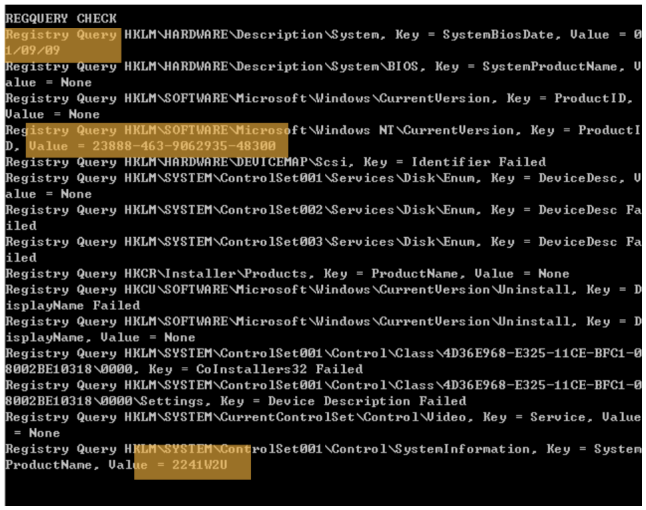

Figure 35: Result of Querying Registries with Original *monitor.dll*

To verify whether modified *monitor.dll* worked or not, I resubmitted *detector.exe* with modified *monitor.dll*. As **Figure 36** and **Figure 37** shown below, the return values of 6 RegOpenKey and 3 RegQueryValue were all obfuscated.

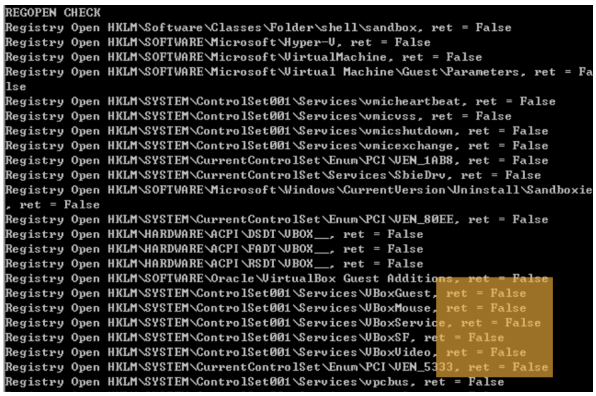

Figure 36: Result of Opening Registries with Modified *monitor.dll*

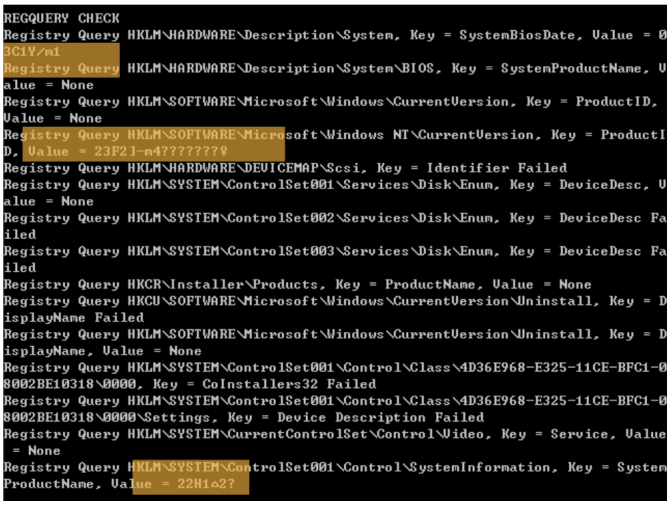

Figure 37: Result of Querying Registries with Modified *monitor.dll*

Besides, to verify whether modified *monitor.dll* worked on real-life malware programs, I submitted a modified *GravityRAT G1*. The reason that using G1 instead of GX was that the operation of GX version depended on its C2 servers, however, its C2 servers were dead, so I could not record any behaviors of GX version. As a result, I modified G1 version with virtual machine and sandbox

evasion techniques appearing in *GravityRAT GX*, like **Figure 38**.

```
private static bool DetectVM(out VirtualMachine virtualMachine)
{
    virtualMachine = default(VirtualMachine);
    bool result = false;
    try
    {
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey("SOFTWARE\\Microsoft\\Virtual Machine\\Guest\\Parameters");
        if (registryKey != null)
        {
            virtualMachine = new VirtualMachine(registryKey.GetValue("HostName").ToString(), registryKey.GetValue
              ("VirtualMachineName").ToString());
            result = true;
        }
    }
```

Figure 38: Evasion Technique Appearing in GX Version

However, evasion techniques appearing in *GravityRAT GX* did not work in the lab's virtual machine environment, I substituted it with other technique, like **Figure 39**.

```
if (Registry.LocalMachine.OpenSubKey("SYSTEM\\ControlSet001\\Services\\VBoxGuest") != null)
{
    Environment.Exit(0);
}
```

Figure 39: Substitute Evasion Technique in G1 Version

Finally, I submitted unmodified *GravityRAT G1*, modified *GravityRAT G1* with original *monitor.dll* and modified *monitor.dll* to Cuckoo Sandbox respectively. With regard to the size of report (**Figure 40**), the size of the first report was almost approximately same as the size of the third report, and the size of the second report had a large extent smaller than the two others. The first report recorded behaviors of unmodified *GravityRAT G1*, and the second report recorded behaviors of modified *GravityRAT G1* with original *monitor.dll*, meanwhile the third report recorded behaviors of modified *GravityRAT G1* with modified *monitor.dll*.

Except for the size, I also observed their behavior, the return value was 2 originally with original *monitor.dll* (**Figure 41**), however, the return value became to be 0 with modified *monitor.dll* (**Figure 41**). This meant the attempted was successful.

```
cuckoo@L307:~/.cuckoo/storage/analyses$ find . -name "report.json" -ls
 21239418 359920 -rw-rw-r--   1 cuckoo   cuckoo   368552864  八  17 16:00 ./1/reports/report.json
 21239395   2168 -rw-rw-r--   1 cuckoo   cuckoo     2216240  八  17 15:58 ./2/reports/report.json
 21239426 359008 -rw-rw-r--   1 cuckoo   cuckoo   367618528  八  17 16:02 ./3/reports/report.json
```

Figure 40: The Size of Reports


```
{
    "category": "registry",
    "status": 1,
    "stacktrace": [],
    "api": "RegOpenKeyExW",
    "return_value": 0,
    "arguments": {
        "access": "0x00020019",
        "base_handle": "0x80000002",
        "key_handle": "0x000001e4",
        "regkey": "HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\Services\\VBoxGuest",
        "regkey_r": "SYSTEM\\ControlSet001\\Services\\VBoxGuest",
        "options": 0
    },
    "time": 1660751910.282,
    "tid": 3680,
    "flags": {}
},
```


Figure 41: The Return Value with Original *monitor.dll*


```
{
    "category": "registry",
    "status": 0,
    "stacktrace": [],
    "last_error": -2146233317,
    "nt_status": 0,
    "api": "RegOpenKeyExW",
    "return_value": 2,
    "arguments": {
        "access": "0x00020019",
        "base_handle": "0x80000002",
        "key_handle": "0x000001e0",
        "regkey": "HKEY_LOCAL_MACHINE\\SYSTEM\\ControlSet001\\Services\\VBoxGuest",
        "regkey_r": "SYSTEM\\ControlSet001\\Services\\VBoxGuest",
        "options": 0
    },
    "time": 1660752014.3595,
    "tid": 4100,
    "flags": {}
},
```


Figure 42: The Return Value with Modified *monitor.dll*

4 Conclusion and Future Work

This research gives a rough review on common evasion techniques used by malware programs. And based on evasion techniques, the preliminary attempt reaches a part of success to prevent malware programs from evading from being detected. To both of *detector.exe* and a part of real-life malware programs, the revision of *monitor.dll* reaches the goal. In addition to *monitor.dll*, the framework prompts analysts to more focus on their main works instead of the aforementioned cumbersome and redundant procedures.

There are still 26 *rst* files and corresponding techniques in *detector.c* should be completed. Not only Registry, but Windows Management Instrumentation is commonly seen in evasion techniques, hence, Windows Management Instrumentation is the next should be completed. In summary, once these anti-evasion techniques are totally revised and the datasets of evasion techniques are enough, the analysis ability of Cuckoo Sandbox will be reinforced and be able to overcome evasion techniques.