Engineering Runtime Malware Detection; Creating a Detection Engine

Joonjae Bang
Carnegie Mellon University
joonjaeb@andrew.cmu.edu

Anusha Penumacha
Carnegie Mellon University
apenumac@andrew.cmu.edu

1 Introduction

Over the years, malware samples have gotten more sophisticated and adept at hiding their activities which triggered a rising need of a malware detection engine. These analysis and detection engines are designed to automate differentiation between malicious and benign processes by observing their behaviors, either in real time or by observing the artifacts left behind by these processes.

In this report, we will discuss the overall construction and operation of our analysis engine, along with our methodologies to detect common malicious behaviors. Next, we will present the results of our analysis engine and a discussion of the results, followed by future directions and potential improvements, and finally a concluding statement.

2 Analysis Engine

The primary operation of our analysis engine is to populate a malware infection tree and determine a suspicion score for the tree based on a series of log files gathered by data collectors running alongside suspicious processes. Based on the final suspicion score calculated by the analysis engine, the analyst can determine whether the original executable that created this infection tree is malicious or not. This section will provide an in depth description of the analysis flow of the engine, the structure of the generated malware infection tree and the suspicion score mechanism.

2.1 Analysis Flow

The analysis engine is designed to iterate over a series of log files to create an infection tree for each log file. It features a configuration manager that takes in a YAML file to determine input log and output file directories, a main analysis engine that parses the log file to build up the infection tree and an output module that saves the results.

The analysis engine itself contains three major components each designed to handle a different type of log entry - file operation logs, registry operation logs and process operation logs. Each component parses their respective logs, evaluates a piece of logic while consulting the current state of the infection tree and updates the infection tree accordingly.

After entirely processing a log file, the analysis engine performs a final post-processing adjustment to the infection tree for signature files, and the final result is written out to a file.

The final infection tree is represented as a Python dictionary structure in JSON format, which allows for easy loading of the infection tree for additional analysis or for graph visualization.

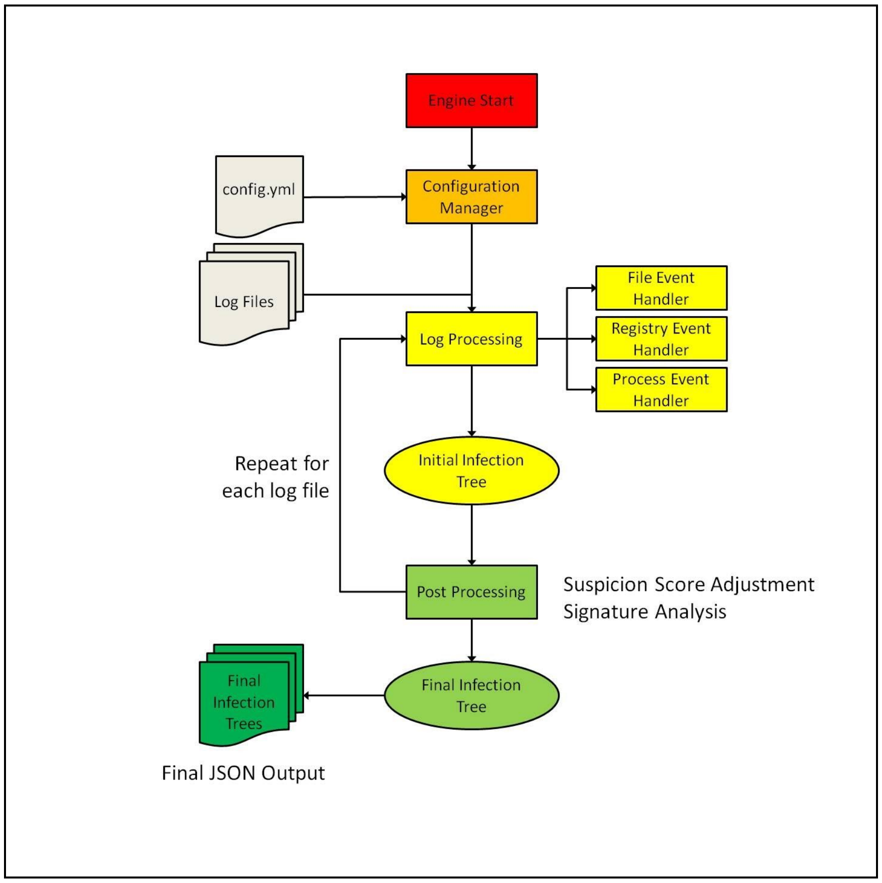


Figure 1. Overall Analysis Flow

2.2 Malware Infection Tree

At the heart of the analysis engine lies the infection tree structure. This tree structure serves to establish a hierarchical relation between the original suspicious process and all its descendants, maintain a collection of system resources that were accessed by these processes and record a list of actions that are related to malware activities. The overall structure of the infection tree is shown in figure 2 below.

In the case of files created, written and deleted by the infection tree, these files are separated according to the specific child process that accessed each file. This separation is a by-product of the internal logic to detect self replication which involves associating each file access with each process. Other system resources such as injected processes and registry keys accessed by these processes are recorded collectively as these system resources contribute to the overall suspicion assessment regardless of which child process performed the access.

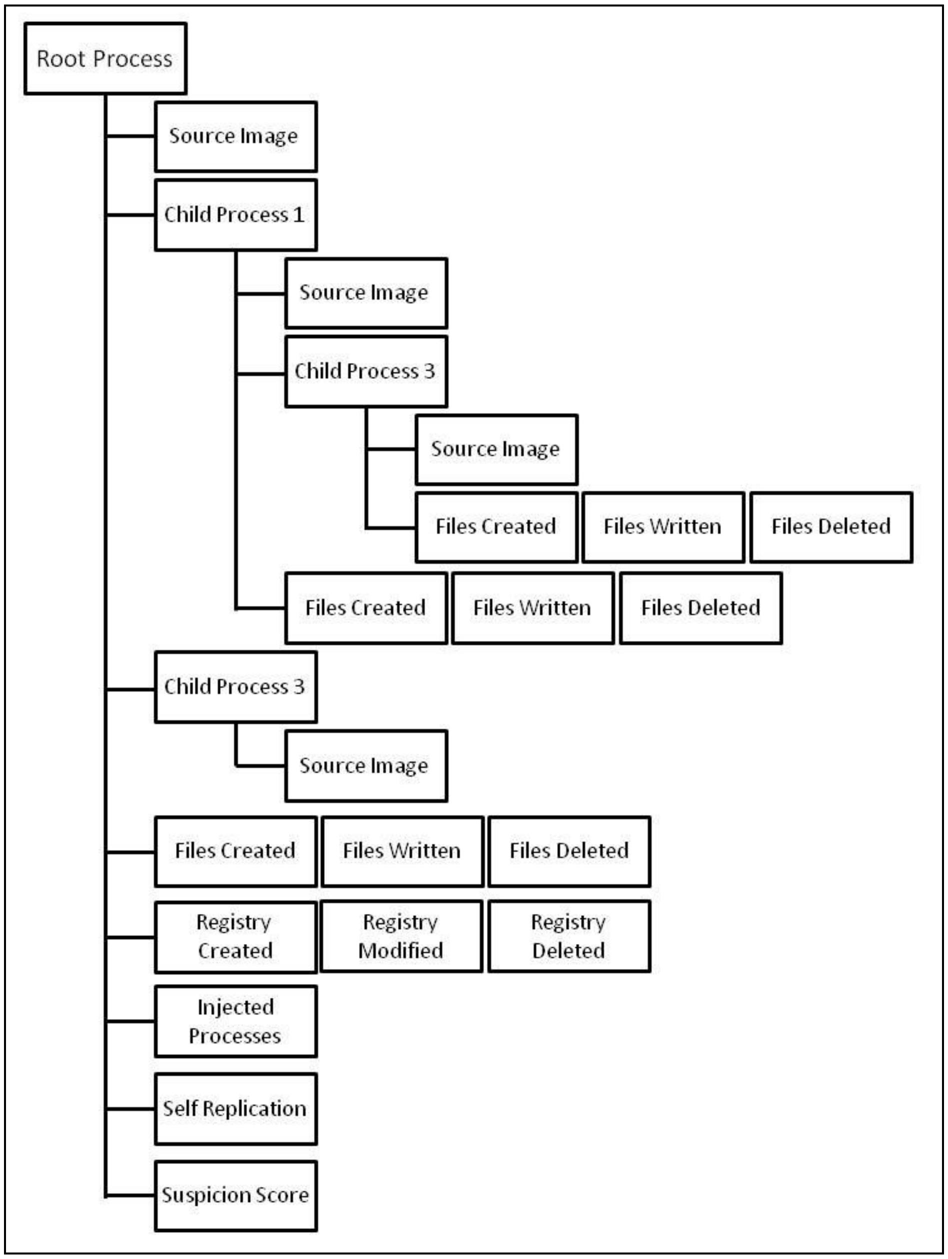


Figure 2. Structure of the Malware Infection Tree

2.3 Suspicion Score

The suspicion score is a metric based on the behavior of the infection tree to determine its maliciousness. Every time a process in this infection tree performs a n action determined by the analysis engine to be potentially malicious, the score of the entire infection tree is increased. This prevents processes from avoiding detection by migrating its malicious functionalities to a child process, since all processes in the tree are judged as a whole.

In the current implementation of the analysis engine, any suspicion score of above 100 is considered malicious.

3 Detection of Specific Malware Behaviour

A primary goal of our analysis engine is to determine the presence of various malware activities that are contained within our logs. This determination is often done by associating a series of different actions, calculating the frequency of a certain action, monitoring system resources that are seen throughout the logs and so on. In this section, we provide a list of malicious activities our analysis engine attempts to detect along with our detection methodology.

3.1 Self Replication

A common behavior among malware is self replication, or copying the whole or part of the malware executable into a different location. The malware executable is usually replicated into an obscure directory location or even into another file, so that even if the original executable is detected and removed the surviving copy can be used to maintain presence on the target machine.

Self replication involves the malware executable loading in its source image into memory, then writing the same source image out to a file. In our log files, these actions are represented as an association of two different logs:

1. An IRP_MJ_READ from the process's source image into one of its memory locations
2. An IRP_MJ_WRITE from the same memory location to another file

Our analysis engine contains the following logic to detect this type of self replication:

1. When a process performs IRP_MJ_READ, the file that is read from along with the memory location is stored in a per-process container that maintains a memory map of what files are loaded into what memory location.
2. When the same process performs IRP_MJ_WRITE, the source address is consulted in the stored memory map. If the filename entry in the memory map corresponds to the process's source image, we detect this action as self replicatory.

This detection methodology is currently limited to detecting self replication on a per-process basis. That is, if a process delegates its self replication to a child process, that replication is not detected with our current mechanism as the analysis engine will not recognize the memory mapped file as the child process's source image.

3.2 Registry and File based Persistence

A primary goal of malware is to maintain persistent access on the target machine even in the case of system reboots, so that the malware remains active for a longer period of time bringing more benefit to its operators. The two most common ways of achieving persistence is to modify special registry keys, and the other is to place the malware executable in special directories that allow the executable to start automatically when the system boots up. The specific registry keys and directories are the following:

Registry

HKLM\\SOFTWARE\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\Run
HKLM\\SOFTWARE\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\RunOnce
HKCU\\[USERNAME]\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
HKCU\\[USERNAME]\\SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce
HKLM\\SYSTEM\\CurrentControlSet\\Services

Directory

\\ProgramData\\Microsoft\\Windows\\Start Menu\\Programs\\Startup

3.3 Excessive Directory Accesses for Ransomware

Ransomware is a type of malware designed to encrypt and prevent access to as many user files as possible. During its encryption process, it often crawls through multiple layers of nested directories searching for files to encrypt.

Our goal is to detect this directory crawling behaviour exhibited by ransomware by monitoring how many unique directories are accessed by a process tree. Whenever a ransomware encrypts a file, it shows up in our log files as an IRP_MJ_WRITE request along with the path of the file being encrypted. We maintain a unique list of directories accessed by a process tree throughout the analysis process. After analyzing the log file, we check the number of entries in our list of directories - if this number is abnormally high, we determine the process tree to belong to a ransomware.

We decided to collectively monitor the directories accessed by the entire process tree rather than monitoring each process separately, in order to handle cases where the ransomware may migrate its encryption functionality to an external process.

3.4 Process Hollowing

One tactic employed by malware to avoid detection is a technique called process hollowing. In process hollowing, a malicious process masquerades itself as a different, unassuming process to make itself less noticeable in monitoring tools such as Process Explorer. The technique involves the malicious process spawning an instance of the unassuming executable in a suspended state, replacing the memory mapped image with its malicious code, and finally resuming the suspended process. The newly created process will run using the unassuming executable’s name and characteristics on the surface, but will be running malicious code underneath much like a wolf in sheep’s clothing.

In order to detect this technique, we make use of the fact that native Windows executables such as explorer.exe or svchost.exe are often chosen as targets of process hollowing; this is because other instances of these executables are often already running in the environment allowing the masquerading process to blend in, and these native Windows executables are reliably present in all Windows environments. With our monitoring tools, process hollowing will can be observed with the following sequence of events:

1. Creation of a native Windows process by a suspicious process
2. Request for a VM Write handle to the newly created Windows process
3. Loading of an image into the newly created Windows process

We can detect these events by adding the following logic to our analysis engine.

1. On load-image, check if the target process is a descendant of our monitored process
2. Check if the source image of the target process is a native Windows executable
3. Check if the image being injected into the process was created by our monitored process

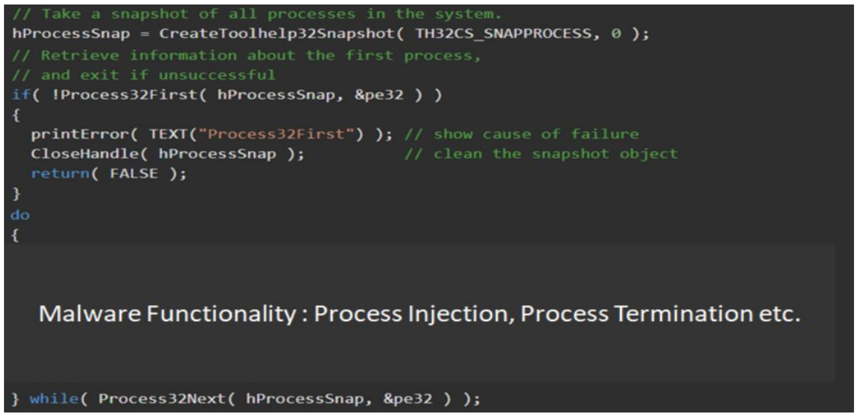
This detection method is mainly limited by the second requirement of the target process being a native Windows executable. Should the malware target a popular non-Windows application or dynamically search for a target application, our analysis engine would not be able to detect these attempts.

3.5 Termination of Non-Descendant Processes

Regular processes rarely terminate processes that are not directly related to the original process. Malware on the other hand often terminates processes outside of its process tree, such as competing malware instances, AV software, monitoring software and more. Thus, any attempts to terminate processes that are outside of the process tree is determined to be suspicious.

3.6 Excessive Process Access Requests

A common behavior among malware is to iterate through a list of running processes in order to inject or terminate them mainly for persistence and stealth purposes. We detect this behaviour by the frequency of consecutive OpenProcess requests detected in our data collectors. We specifically limit detection to OpenProcess requests with VM Write or Terminate access codes, as other types of accesses are also used often by other processes. The consecutive requirement follows common programming conventions in malware for iterating processes, as shown in Figure 3.



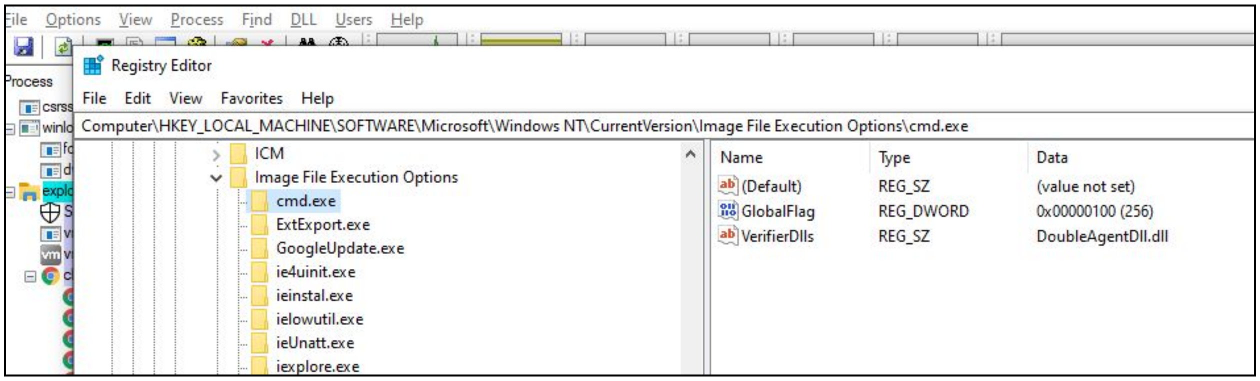
```
// Take a snapshot of all processes in the system.
hProcessSnap = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );
// Retrieve information about the first process,
// and exit if unsuccessful
if( !Process32First( hProcessSnap, &pe32 ) )
{
  printError( TEXT("Process32First") ); // show cause of failure
  CloseHandle( hProcessSnap );          // clean the snapshot object
  return( FALSE );
}
do
{

    Malware Functionality : Process Injection, Process Termination etc.

} while( Process32Next( hProcessSnap, &pe32 ) );
```

Figure 3. Common Practice for Iterating through Running Processes

3.7 DLL Injection through the Image File Execution Options Registry

Microsoft offers a standard way to install runtime verification tools for native code via Microsoft Application Verifier Provider DLLs. A verifier provider DLL is simply a DLL that is loaded into the process and is responsible for performing runtime verifications for the application.

In order to register a new Application Verifier Provider DLL one needs to create a verifier provider DLL and register it by creating a set of keys in the registry. Once a DLL has been registered as a verifier provider DLL for a process, it would permanently be injected by the Windows Loader into the process every time the process starts, even after reboots/updates/reinstalls/patches/etc. So we can create a registry key with any executable name and add a verified DLL's subkey under the key which points to our custom DLL.


File Options View Process Find DLL Users Help
Registry Editor
File Edit View Favorites Help

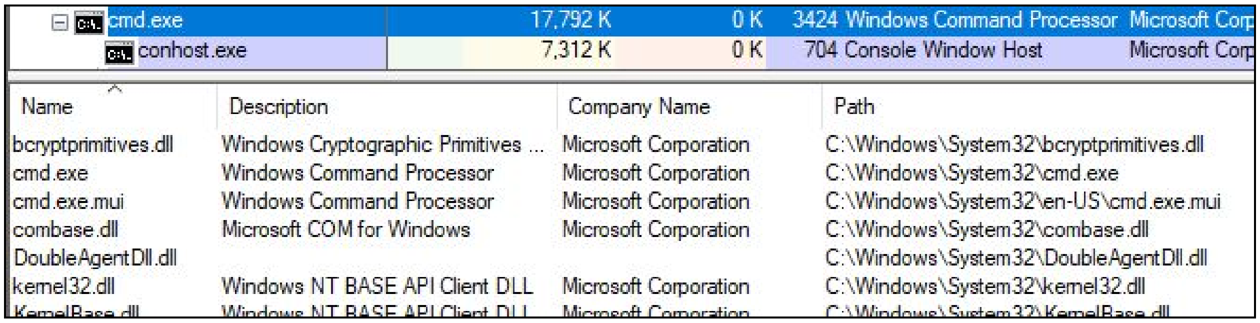
Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\cmd.exe

```
ICM
Image File Execution Options
    cmd.exe
    ExtExport.exe
    GoogleUpdate.exe
    ie4uinit.exe
    ieinstal.exe
    ielowutil.exe
    ieUnatt.exe
    iexplore.exe
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
GlobalFlag	REG_DWORD	0x00000100 (256)
VerifierDlls	REG_SZ	DoubleAgentDll.dll

Figure 4. Adding a Rogue DLL to Image File Execution Options

As shown in the screenshot above, now DoubleAgentDll.dll is a verified dll for cmd.exe. Whenever cmd.exe is launched, the DLL will be injected into the process



```
cmd.exe        17,792 K   0 K   3424 Windows Command Processor  Microsoft Corp
  conhost.exe   7,312 K   0 K    704 Console Window Host         Microsoft Corp
```

Name	Description	Company Name	Path
bcryptprimitives.dll	Windows Cryptographic Primitives ...	Microsoft Corporation	C:\Windows\System32\bcryptprimitives.dll
cmd.exe	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\cmd.exe
cmd.exe.mui	Windows Command Processor	Microsoft Corporation	C:\Windows\System32\en-US\cmd.exe.mui
combase.dll	Microsoft COM for Windows	Microsoft Corporation	C:\Windows\System32\combase.dll
DoubleAgentDll.dll			C:\Windows\System32\DoubleAgentDll.dll
kernel32.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\System32\kernel32.dll
KernelBase.dll	Windows NT BASE API Client DLL	Microsoft Corporation	C:\Windows\System32\KernelBase.dll

Figure 5. Malicious DLL loaded into Target Executable

The injection will persist even after system reboot. This way an attacker can simultaneously achieve code injection and persistence. This mode of injection is often more difficult to detect because a direct correlation cannot be established between an attacker and the victim process. This exploits a 15 years old legitimate feature of windows and therefore cannot be patched.

Our analysis engine detects this in the following manner. Our data collectors present information for every registry modification. We can look for any modification [creation/addition/deletion] to the Image File Execution Options Registry key.

If a new process name is added under IFEO registry, we can create a strong bond between the process that is adding the entry and the process for which the entry is being added for. We can start monitoring the new process. The malicious DLL injection will look legitimate in the new process.

Additional features: If we can get the value of the DLLs that are being added to the VerifiedDLLs key, we can determine the malicious DLL name that is being injected into the victim process.

Example output : In the output below, f4 is the new strong bond between the attacker and the victim process

```
--------------------------------------------------
\??\c:\test\DoubleAgent_x64.exe-->DoubleAgentDll.dll : f3,f3
\??\c:\test\DoubleAgent_x64.exe-->sleep_project.exe : f4
\??\c:\test\DoubleAgent_x64.exe-->\REGISTRY\MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options\sleep_project.exe : r2
\??\c:\test\DoubleAgent_x64.exe with pid 4592 has an overall score of 100.0
```

4 Results

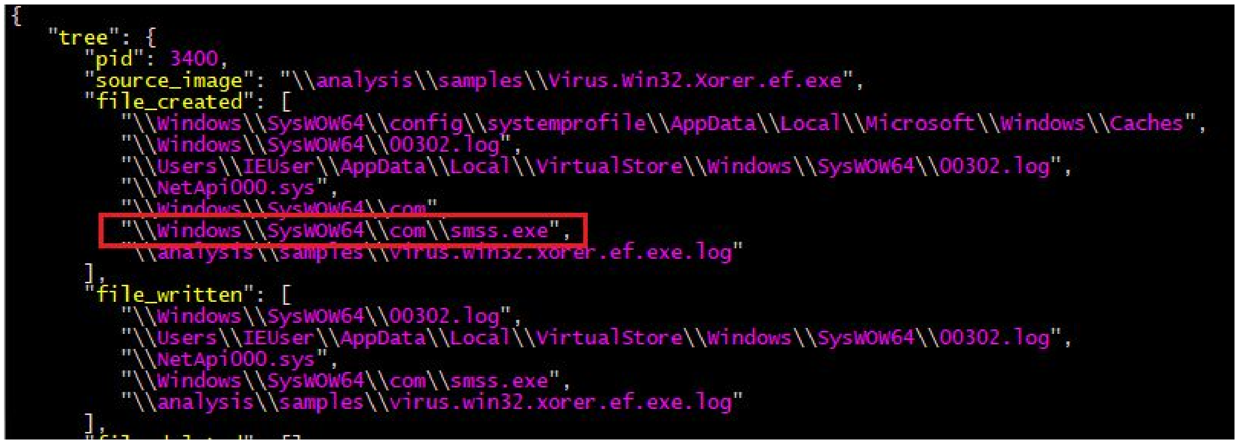
In this section, we present some sample analysis results for both malware and benign software, and discuss various actions detected through our analysis engine along with a breakdown in suspicion scores.

4.1 Xorer.ef.exe

Xorer is a malware sample that appears to perform basic self replication and spawns multiple child processes. Many of these child processes are native Windows executables such as cmd.exe, cacls.exe and Conhost.exe, and are most likely used to perform various malware functionalities that our engine did not detect. Furthermore, Xorer also creates two executables with the same name as native Windows executables, smss.exe and lsass.exe, but under a different directory under Windows. Its overall suspicion score is 120570.0, and most of this can be attributed to multiple creations of smss.exe and lsass.exe in a privileged directory.

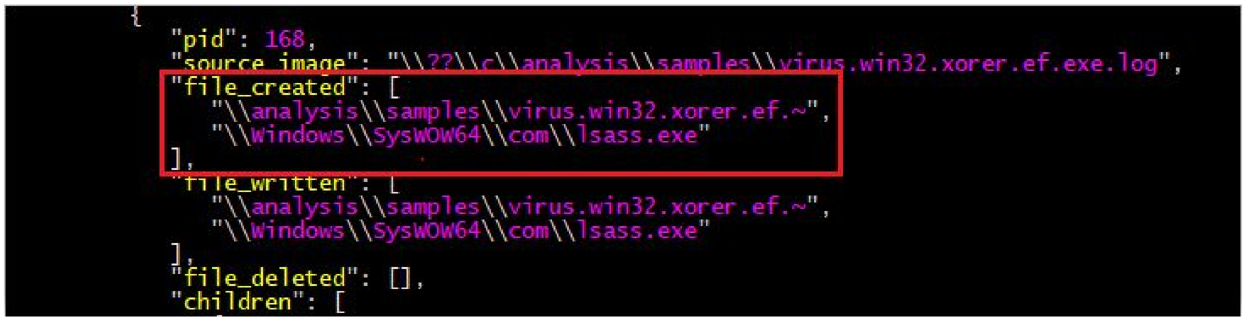
Overall, these are the most significant activities that led to an increased suspicion score:

- Modification of privileged Windows directories
- Multiple attempts at self replication
- Modifications of dangerous registry values



```
{
   "tree": {
      "pid": 3400,
      "source_image": "\\analysis\\samples\\Virus.Win32.Xorer.ef.exe",
      "file_created": [
         "\\Windows\\SysWOW64\\config\\systemprofile\\AppData\\Local\\Microsoft\\Windows\\Caches",
         "\\Windows\\SysWOW64\\00302.log",
         "\\Users\\IEUser\\AppData\\Local\\VirtualStore\\Windows\\SysWOW64\\00302.log",
         "\\NetApi000.sys",
         "\\Windows\\SysWOW64\\com",
         "\\Windows\\SysWOW64\\com\\smss.exe",
         "\\analysis\\samples\\Virus.Win32.Xorer.ef.exe.log"
      ],
      "file_written": [
         "\\Windows\\SysWOW64\\00302.log",
         "\\Users\\IEUser\\AppData\\Local\\VirtualStore\\Windows\\SysWOW64\\00302.log",
         "\\NetApi000.sys",
         "\\Windows\\SysWOW64\\com\\smss.exe",
         "\\analysis\\samples\\virus.win32.xorer.ef.exe.log"
      ],
```

Figure 6. Creation of fake smss.exe by root instance of Xorer.ef.exe



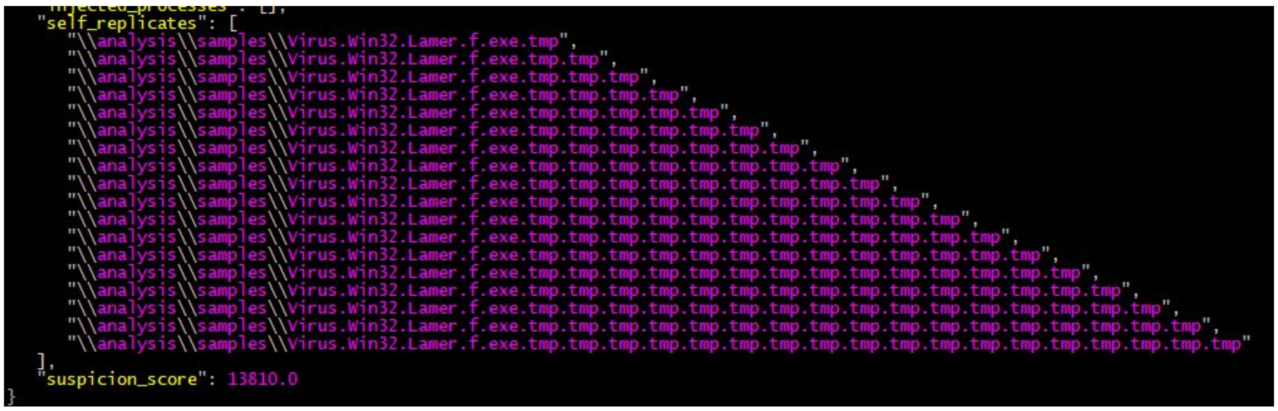
```
         {
            "pid": 168,
            "source_image": "\\??\\c\\analysis\\samples\\virus.win32.xorer.ef.exe.log",
            "file_created": [
               "\\analysis\\samples\\virus.win32.xorer.ef.~",
               "\\Windows\\SysWOW64\\com\\lsass.exe"
            ],
            "file_written": [
               "\\analysis\\samples\\virus.win32.xorer.ef.~",
               "\\Windows\\SysWOW64\\com\\lsass.exe"
            ],
            "file_deleted": [],
            "children": [
```

Figure 7. Creation of fake lsass.exe by a descendant instance of Xorer.ef.exe

4.2 Lamer.f.exe

Lamer.f.exe is another sample that performs multiple self replications and starts new instances of itself from these self replicated files. Every time it self replicates, the new replicated file adds a ".tmp" extension to the original filename leading to a copy that has multiple .tmp extensions appended to the end of the original filename.

The suspicion score of this sample is 13810.0, and it's score is mainly attributed to this repeated self replication attempts. It does not appear to create, modify or delete any other files and registries.



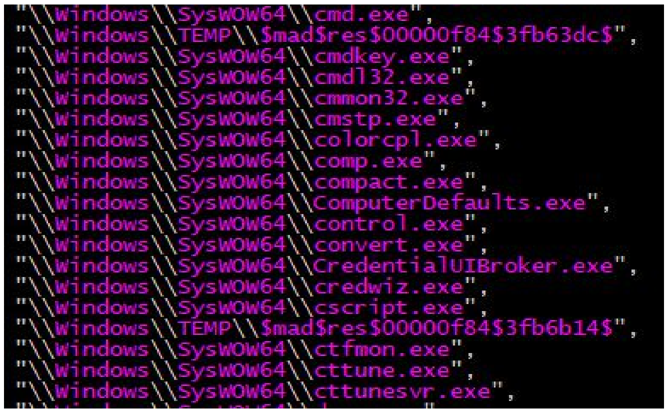
```
    "injected_processes": [],
    "self_replicates": [
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp",
        "\\analysis\\samples\\Virus.Win32.Lamer.f.exe.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp.tmp"
    ],
    "suspicion_score": 13810.0
}
```

Figure 8. Lamer.f.exe's Self Replication

4.3 Lamer.i.exe

Lamer.i.exe is a malware sample that attempts to heavily modify the default Windows executables. It creates and modifies multiple files and executables in the C:\Windows and C:\Windows\SysWOW64 directory, such as notepad.exe, nslookup.exe and more. On the surface it appears to be a type of virus that attaches itself to files, but we have not detected any self replication attempts. It may be that this malware performs partial self replication, which the engine cannot detect at the moment.

The malware sample has a suspicion score of 160910.0, and a significant majority is attributed to this infection of executables in privileged Windows directories.



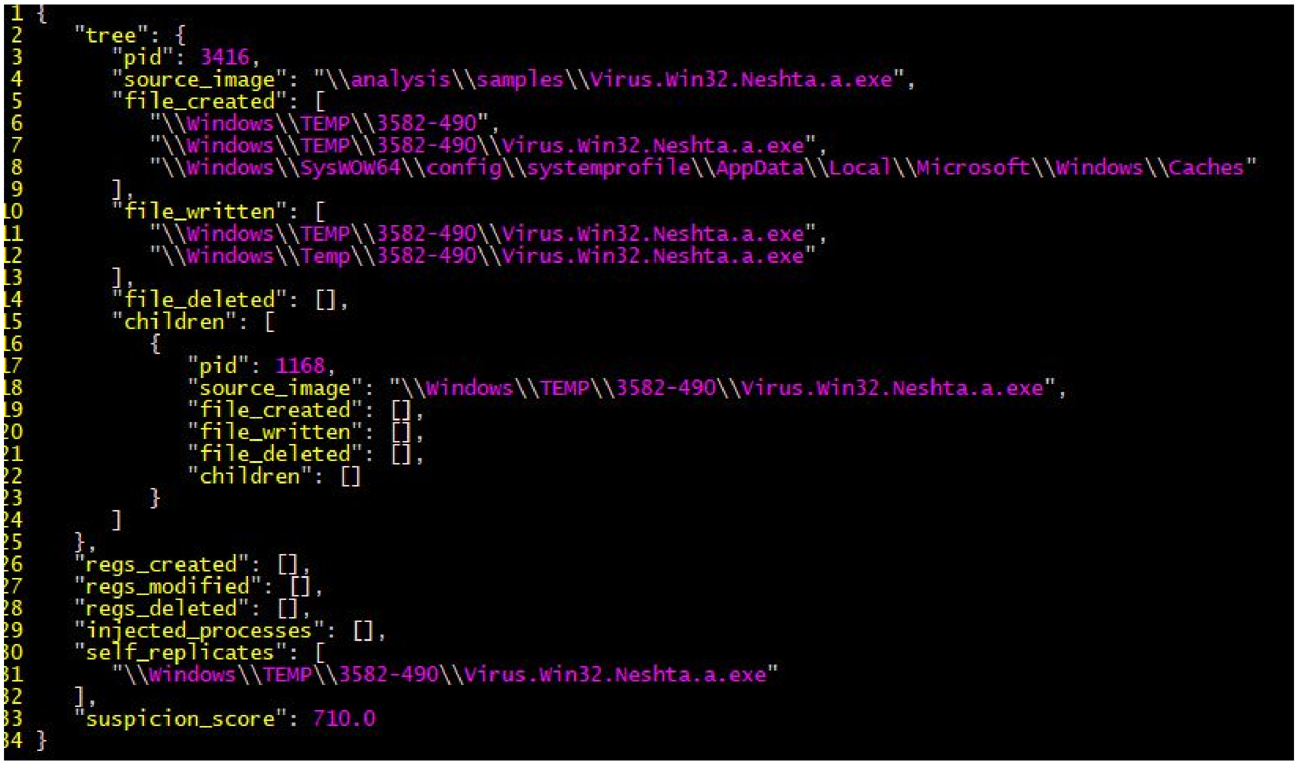
```
"\\Windows\\SysWOW64\\cmd.exe",
"\\Windows\\TEMP\\$mad$res$00000f84$3fb63dc$",
"\\Windows\\SysWOW64\\cmdkey.exe",
"\\Windows\\SysWOW64\\cmdl32.exe",
"\\Windows\\SysWOW64\\cmmon32.exe",
"\\Windows\\SysWOW64\\cmstp.exe",
"\\Windows\\SysWOW64\\colorcpl.exe",
"\\Windows\\SysWOW64\\comp.exe",
"\\Windows\\SysWOW64\\compact.exe",
"\\Windows\\SysWOW64\\ComputerDefaults.exe",
"\\Windows\\SysWOW64\\control.exe",
"\\Windows\\SysWOW64\\convert.exe",
"\\Windows\\SysWOW64\\CredentialUIBroker.exe",
"\\Windows\\SysWOW64\\credwiz.exe",
"\\Windows\\SysWOW64\\cscript.exe",
"\\Windows\\TEMP\\$mad$res$00000f84$3fb6b14$",
"\\Windows\\SysWOW64\\ctfmon.exe",
"\\Windows\\SysWOW64\\cttune.exe",
"\\Windows\\SysWOW64\\cttunesvr.exe",
```

Figure 9. Partial List of Executables Modified by Lamer.i.exe

4.4 Neshta.a.exe

Like other samples in this set, Neshta.a.exe also performs self replication, spawns multiple instances of itself and modifies a privileged Windows directory. As with Xorer.ef.exe, it creates a single copy of itself that it overwrites repeatedly, instead of creating a new copy of itself with every self replication.

Its suspicion score is 710.0, and it can be attributed to multiple self replication attempts and writing to a C:\Windows\SysWOW64 directory.



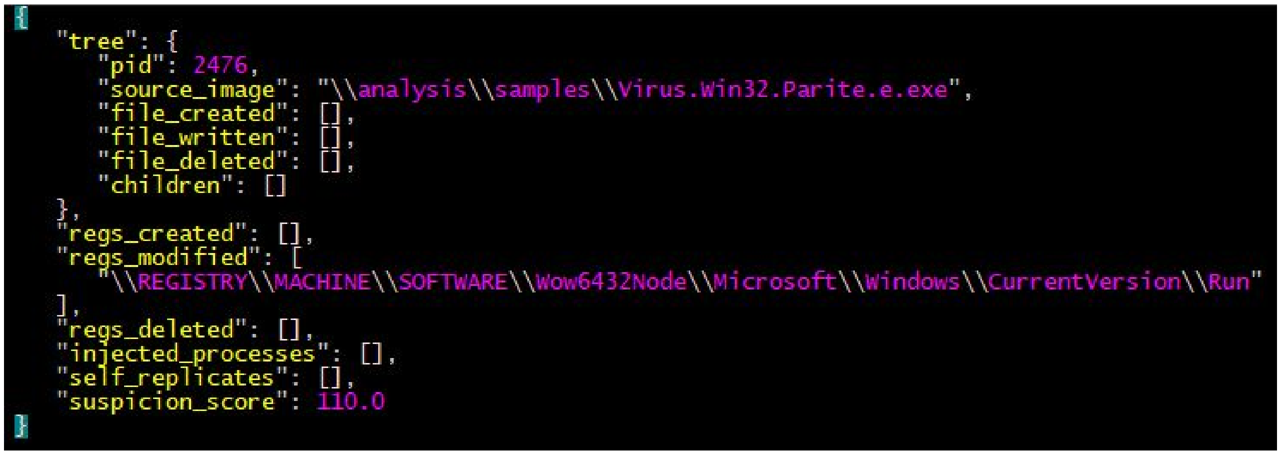
```
{
   "tree": {
      "pid": 3416,
      "source_image": "\\analysis\\samples\\Virus.Win32.Neshta.a.exe",
      "file_created": [
         "\\Windows\\TEMP\\3582-490",
         "\\Windows\\TEMP\\3582-490\\Virus.Win32.Neshta.a.exe",
         "\\Windows\\SysWOW64\\config\\systemprofile\\AppData\\Local\\Microsoft\\Windows\\Caches"
      ],
      "file_written": [
         "\\Windows\\TEMP\\3582-490\\Virus.Win32.Neshta.a.exe",
         "\\Windows\\Temp\\3582-490\\Virus.Win32.Neshta.a.exe"
      ],
      "file_deleted": [],
      "children": [
         {
            "pid": 1168,
            "source_image": "\\Windows\\TEMP\\3582-490\\Virus.Win32.Neshta.a.exe",
            "file_created": [],
            "file_written": [],
            "file_deleted": [],
            "children": []
         }
      ]
   },
   "regs_created": [],
   "regs_modified": [],
   "regs_deleted": [],
   "injected_processes": [],
   "self_replicates": [
      "\\Windows\\TEMP\\3582-490\\Virus.Win32.Neshta.a.exe"
   ],
   "suspicion_score": 710.0
}
```

Figure 10. Malware Infection Tree of Neshta.a.exe

4.5 Parite.e.exe

Parity.e.exe only modifies the CurrentVersion\Run registry for persistence. The engine has not detected any other files or registries that are created, modified or deleted by this malware, nor does it perform any process injection to external processes.

Its suspicion score is 110.0, attributable to its attempts at registry based persistence.



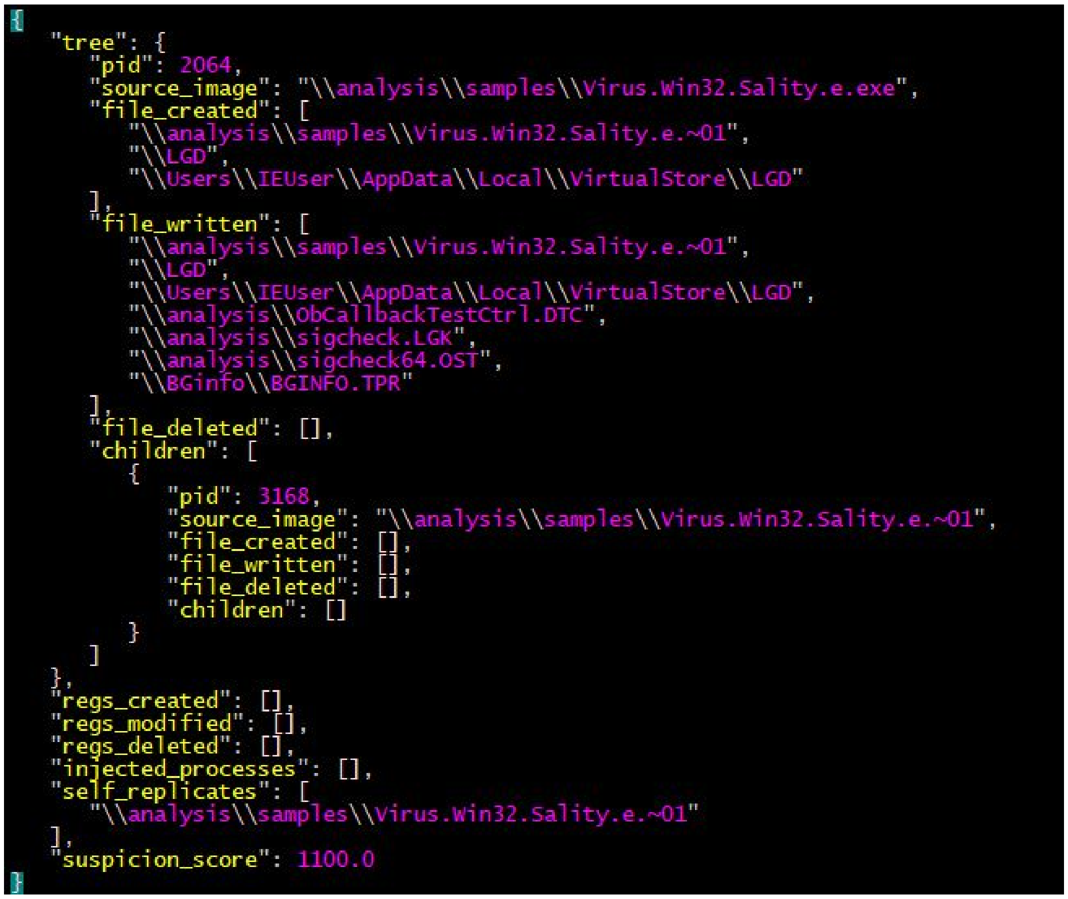
```
{
   "tree": {
      "pid": 2476,
      "source_image": "\\analysis\\samples\\Virus.Win32.Parite.e.exe",
      "file_created": [],
      "file_written": [],
      "file_deleted": [],
      "children": []
   },
   "regs_created": [],
   "regs_modified": [
      "\\REGISTRY\\MACHINE\\SOFTWARE\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\Run"
   ],
   "regs_deleted": [],
   "injected_processes": [],
   "self_replicates": [],
   "suspicion_score": 110.0
}
```

Figure 11. Malware Infection Tree of Parite.e.exe

4.6 Sality.e.exe

Like other malware samples, Sality.e.exe performs self replication and spawns a new child process from its copy. Like other samples on this list, it creates a single copy that is overwritten repeatedly. It also creates and modifies several files, but none are located in privileged directories and their purposes are not clear.

The malware has a suspicion score of 1100.0, and it is fully attributable to its repeated attempts at self replication.



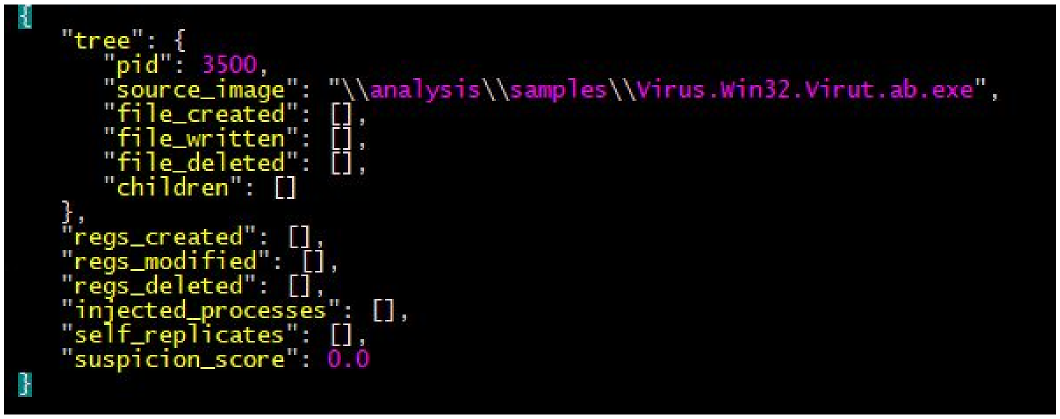
```
{
   "tree": {
      "pid": 2064,
      "source_image": "\\analysis\\samples\\Virus.Win32.Sality.e.exe",
      "file_created": [
         "\\analysis\\samples\\Virus.Win32.Sality.e.~01",
         "\\LGD",
         "\\Users\\IEUser\\AppData\\Local\\VirtualStore\\LGD"
      ],
      "file_written": [
         "\\analysis\\samples\\Virus.Win32.Sality.e.~01",
         "\\LGD",
         "\\Users\\IEUser\\AppData\\Local\\VirtualStore\\LGD",
         "\\analysis\\ObCallbackTestCtrl.DTC",
         "\\analysis\\sigcheck.LGK",
         "\\analysis\\sigcheck64.OST",
         "\\BGinfo\\BGINFO.TPR"
      ],
      "file_deleted": [],
      "children": [
         {
            "pid": 3168,
            "source_image": "\\analysis\\samples\\Virus.Win32.Sality.e.~01",
            "file_created": [],
            "file_written": [],
            "file_deleted": [],
            "children": []
         }
      ]
   },
   "regs_created": [],
   "regs_modified": [],
   "regs_deleted": [],
   "injected_processes": [],
   "self_replicates": [
      "\\analysis\\samples\\Virus.Win32.Sality.e.~01"
   ],
   "suspicion_score": 1100.0
}
```

Figure 12. Malware Infection Tree of Sality.e.exe

4.7 Dormant Malware Samples

During our analysis, we discovered two malware samples that were completely dormant during the monitoring process. These were Downloader.av.exe and Virut.ab.exe, both of which did not create, modify or delete any system resources, nor were they detected performing any type of injection into an external process. This could be because the sample has some method of detecting a monitoring environment, crashed during its execution or simply does nothing significant at the moment.

As expected these samples received a suspicion score of 0.0.



```
{
    "tree": {
        "pid": 3500,
        "source_image": "\\analysis\\samples\\Virus.Win32.Virut.ab.exe",
        "file_created": [],
        "file_written": [],
        "file_deleted": [],
        "children": []
    },
    "regs_created": [],
    "regs_modified": [],
    "regs_deleted": [],
    "injected_processes": [],
    "self_replicates": [],
    "suspicion_score": 0.0
}
```

Figure 13. Malware Infection Tree of Dormant Sample

4.8 False Positives on Benign Samples

During our analysis, the engine detected two benign samples as being potentially malicious - BitTorrent.exe and iTunes64Setup.exe. The two executables were heavily involved with modifying the privileged Windows directory and set up the same registry keys that are associated with malware persistence, most likely to enable these software to automatically start whenever the computer is booted.

However, compared to other malware samples their suspicion scores are significantly lower, as they do not modify privileged directories beyond what is required for their functionality. We believe that we can handle these false positives through a better scoring scheme based on more empirical data of common benign software activities.

BitTorrent.exe had a suspicion score of 296.0, mainly attributable to the modification of registry keys used for persistence. iTunes64Setup.exe had a score of 140.0, mainly attributable to modifying privileged directories.

5 Future Direction

While the analysis engine accomplishes our original goals, there are still many components that need improvements, both in the data collection and analysis component. In this section we lay out some of these improvements and how they can help the analysis engine overcome current limitations.

5.1 Additional IRP_MJ_READ Information

Currently when the data collectors log IRP_MJ_READ requests, they fail to log the total size of data that is requested to be read in, only the destination address. This places a limitation on our capability to detect partial self replication, as we only know the address where the source image is loaded instead of a wider memory range. Having this IRP_MJ_READ size information will allow the analysis engine to monitor a section of memory instead of a single address for subsequent IRP_MJ_WRITE requests to detect attempts at partial self replication.

5.2 Monitoring Injected Processes

When the suspicious process injects malicious code into processes outside its infection tree, the analysis engine only detects the action of process injection. However, it does not currently monitor the actions of the injected processes. Adding these processes on the list of processes to be monitored can provide further insight into the capabilities of the malware sample under analysis.

5.3 Detection of Native Windows Executables as Malicious

As seen in some of our analysis results, many executables that closely interact with the Windows environment are also detected as potentially malicious by our analysis engine. This is because many actions performed by these executables can also easily be abused by malware to achieve nefarious purposes. The analysis engine will require more sophisticated logic, more extensive data collector logs and a whitelist of trusted executables in order to better distinguish legitimate executables from malicious ones.

5.4 Adjusted Suspicion Scoring Mechanisms

The current scoring mechanism is rather basic, assigning static values for specific actions and regarding any process tree of above 100 points as malicious. The analysis engine needs more insight into the behaviors of both malicious and benign processes, and require further adjustments to the scoring system both in terms of how many points are given for a specific action and what the threshold for malicious processes should be.

6 Conclusion

Our analysis engine currently detects various basic malware functionalities such as self replication, process injection and few registry Persistence mechanisms. We have also demonstrated that these detection mechanisms can be extended to recognize more advanced forms of malware behavior such as ransomware encryption and DLL injection through IFEO registry modification. In order to extend the analysis engine to detect more sophisticated forms of malware behavior, we require various improvements in both the data collection mechanisms and our analysis logic.