See Ya Sharp: A Loader's Tale

mcafee.com/blogs/enterprise/mcafee-enterprise-atr/see-ya-sharp-a-loaders-tale/

August 4, 2021

By Max Kersten on Aug 04, 2021

Introduction

The DotNet based CyaX-Sharp loader, also known as ReZer0, is known to spread commodity malware, such as AgentTesla. In recent years, this loader has been referenced numerous times, as it was used in campaigns across the globe. The tale of CyaX-Sharp is interesting, as the takeaways provide insight into the way actors prefer to use the loader. Additionally, it shines a light onto a spot that is not often illuminated: the inner workings of loaders.

This blog is split up into several segments, starting with a brief preface regarding the coverage of loaders in reports. After that, the origin of the loader's name is explored. Next, the loader's capabilities are discussed, as well as the automatic extraction of the embedded payload from the loader. Lastly, the bulk analysis of 513 unique loader samples is discussed.

Loaders and their Coverage in Blogs

To conceal the malware, actors often use a loader. The purpose of a loader is, as its name implies, to load and launch its payload, thereby starting the next stage in the process. There can be multiple loaders that are executed sequentially, much like a Russian Matryoshka doll in which the smallest doll, which is hidden inside numerous others, is the final payload. The

“smallest doll” generally contains the malware’s main capabilities, such as stealing credentials, encrypting files, or providing remote access to the actor.

While there is a lot of research into the actions of the final payload, the earlier stages are just as interesting and relevant. Even though the earlier stages do not contain the capabilities of the malware that is eventually loaded, they provide insight as to what steps are taken to conceal the malware. Blogs generally mention the capabilities of a loader briefly, if at all. The downside here lies in the potential detection rules that others can create with the blog, as the focus is on the final step in the process, whereas the detection should start as soon as possible.

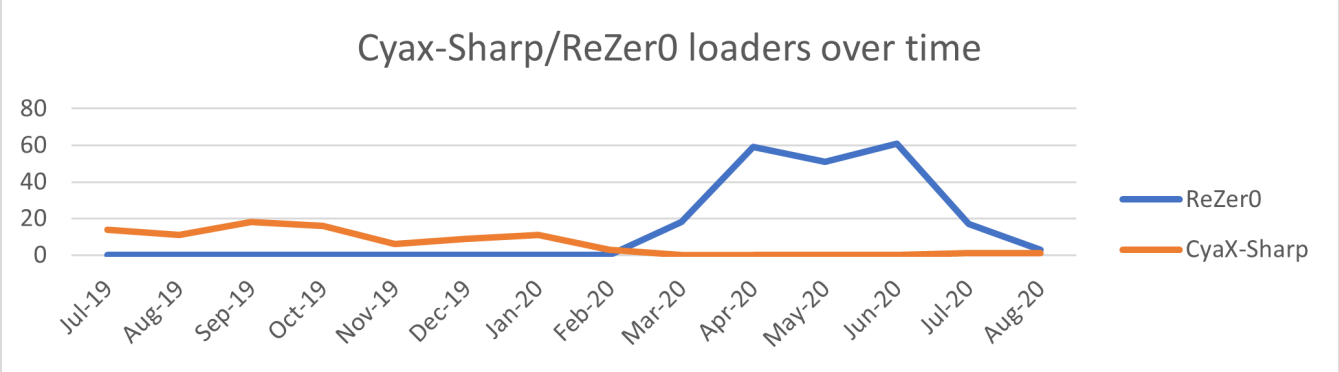
Per best security practices, organizations should protect themselves at every step along the way, rather than only focusing on the outside perimeter. These threat models are often referred to as the, respectively, onion and egg model. The egg’s hard shell is tough to break, but once inside, an attacker has free roam. The onion model opposes the attacker every step of the way, due to its layered approach. Knowing the behavior of the final payload is helpful to detect and block malware although, ideally, the malware would be detected as early on as possible.

This blog focuses on one specific loader family, but the takeaways are valid in a broader sense. The preferred configurations of the actors are useful to understand how loaders can be used in a variety of attacks.

Confusing Family Names

A recent blog by G Data’s Karsten Hahn provides a more in-depth look into malware families ambiguous naming schemes. This loader’s name is also ambiguous, as it is known by several names. Samples are often named based on distinctive characteristics in them. The name CyaX-Sharp is based upon the recurring string in samples. This is, however, exactly why it was also named ReZer0.

When looking at the most used names within the 513 obtained samples, 92 use CyaX-Sharp, whereas 215 use ReZer0. This would make it likely that the loader would be dubbed ReZer0, rather than CyaX-Sharp. However, when looking at the sample names over time, as can be seen in the graph below, the reason why CyaX-Sharp was chosen becomes apparent: the name ReZer0 was only introduced 8 months after the first CyaX-Sharp sample was discovered. Based on this, McAfee refers to this loader as CyaX-Sharp.



Within the settings, one will find *V2* or *V4*. This is not a reference of the loader’s version, but rather the targeted DotNet Framework version. Within the sample set, 62% of the samples are compiled to run on V4, leaving 38% to run on V2.

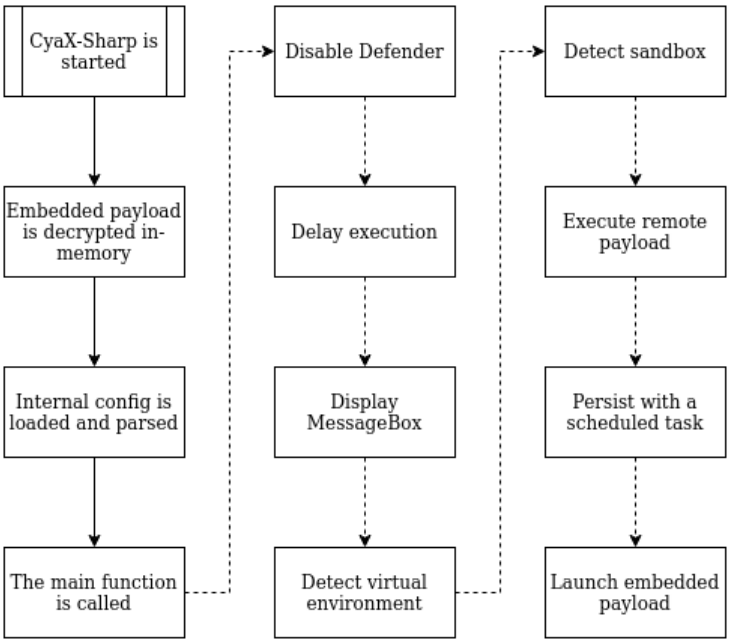
The Loader’s Capabilities

Each version of the loader contains all core capabilities, which may or may not be executed during runtime, based on the loader’s configuration. The raw configurations are stored in a string, using two pipes as the delimiting value. The string is then converted into a string array using said delimiter. Based on the values at specific indices, certain capabilities are enabled. The screenshots below show, respectively, the raw configuration value, and some of the used indices in a sample (SHA-256: a15be1bd758d3cb61928ced6cdb1b9fa39643d2db272909037d5426748f3e7a4).

```
// Token: 0x04000003 RID: 3
private static string k = "3||1||1||0||0|||||0||0||0||0|||||||||||||0||0||0||0||0||0||0||0||v4||0||2976||";

// Token: 0x04000004 RID: 4
private static string[] aa = Strings.Split(X.k, "||", -1, CompareMethod.Binary);

// Token: 0x04000005 RID: 5
private static int InjectValue = Conversions.ToInteger(X.aa[0]);

// Token: 0x04000006 RID: 6
private static int isStartup = Conversions.ToInteger(X.aa[1]);
```

```
string location = Assembly.GetEntryAssembly().Location;
Random random = new Random();
Thread.Sleep(random.Next(45000, 60000));
bool flag = X.DetectGawadaka();
if (flag)
{
    Environment.Exit(0);
}
bool flag2 = X.AntiEm == 1;
if (flag2)
{
    WinDefender.Disable();
}
bool flag3 = X.AntiVm == 1;
if (flag3)
{
    bool flag4 = Antis.AntiVM();
    if (flag4)
    {
        Environment.Exit(0);
    }
}
bool flag5 = X.AntiS8 == 1;
if (flag5)
{
    bool flag6 = Antis.AntiS8(location);
    if (flag6)
    {
        Environment.Exit(0);
    }
}
```

The loader can delay its execution by sleeping for a certain number of seconds, use a mutex to ensure it is not already running, display a message box with a custom message, persist itself as a scheduled task, and/or execute a given payload in several ways. The payload can be downloaded from an external location, after which it is started. Alternatively, or additionally, the embedded payload within the loader can be launched. This can be done directly from the loader's memory with the help of reflective calls, or by hollowing a newly created process. The flowchart below visualizes the process. Note that the dotted line means the linked step can be skipped, depending on the loader's configuration.



Process Hollowing

The newly created process is one of the following: MSBuild.exe, vbc.exe, RegSvcs.exe, or a new instance of the loader. The process hollowing code segment seems to be taken from NYAN-x-CAT's GitHub, as the for-loop to start the process hollowing method is present in both the loader and the linked repository. The way an error is handled is not a standardized method,

making the link between the publicly available code very likely. The first image below shows the original code from the repository, whereas the second image shows the code from the loader (SHA-256: a15be1bd758d3cb61928ced6cdb1b9fa39643d2db272909037d5426748f3e7a4)

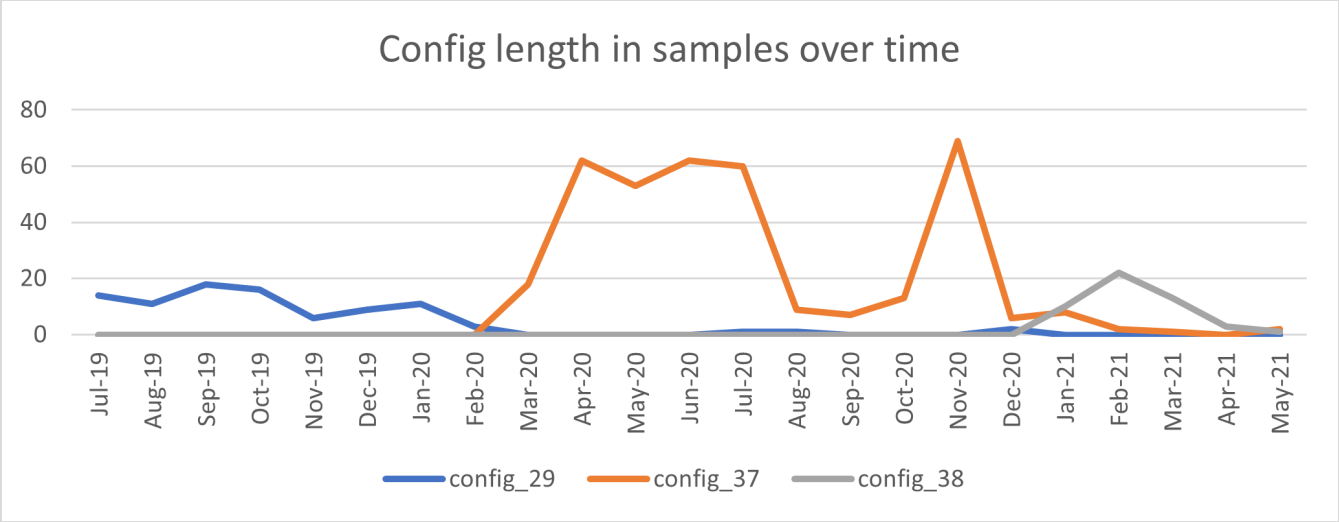
```
0 references
public static void Execute(string path, byte[] payload)
{
    for (int i = 0; i < 5; i++)
    {
        int readWrite = 0x0;
        StartupInformation si = new StartupInformation();
        ProcessInformation pi = new ProcessInformation();
        si.Size = Convert.ToUInt32(Marshal.SizeOf(typeof(StartupInformation)));
```

```
// Token: 0x0600001A RID: 26 RVA: 0x000021C0 File Offset: 0x000003C0
public static bool Run(string path, byte[] data)
{
    int num = 1;
    checked
    {
        for (;;)
        {
            bool flag = Bro.HandleRun(path, string.Empty, data, false);
            if (flag)
            {
                break;
            }
            num++;
            if (num > 5)
            {
                goto Block_2;
            }
        }
        return true;
        Block_2:
        return false;
    }
}
```

The loop calls the process hollowing function several times to more easily handle exceptions. In the case of an exception during the process hollowing, the targeted process is killed and the function returns. To try several times, a loop is used.

Changes Over Time

Even though the loader has changed over time, it maintained the same core structure. Later versions introduced minor changes to existing features. Below, different loader versions will be described, where the length of the string array that contains the loader's configuration is used to identify different versions. The graph shows the rise and fall for each of the versions.



There are two notable differences in versions where the config array's size is larger than 29. Some specific samples have slightly different code when compared with others, but I did not consider these differences sizable enough to warrant a new version.

Firstly, the ability to enable or disable the delayed execution of a sample. If enabled, the execution is delayed by sleeping for a predefined number of seconds. In config_29, the delay functionality is always enabled. The duration of the delay is based on the `System.Random` object, which is instantiated using the default seed. The given lower and upper limits are 45,000 and 60,000, resulting in a value between these limits, which equals in the number of milliseconds the execution should be delayed.

Secondly, the feature to display a custom message in a prompt has been added. The config file contains the message box' title, text, button style, and icon style. Prompts can be used to display a fake error message to the victim, which will appear to be legitimate e.g. 43d334c125968f73b71f3e9f15f96911a94e590c80955e0612a297c4a792ca07, which uses "You do not have the proper software to view this document" as its message.

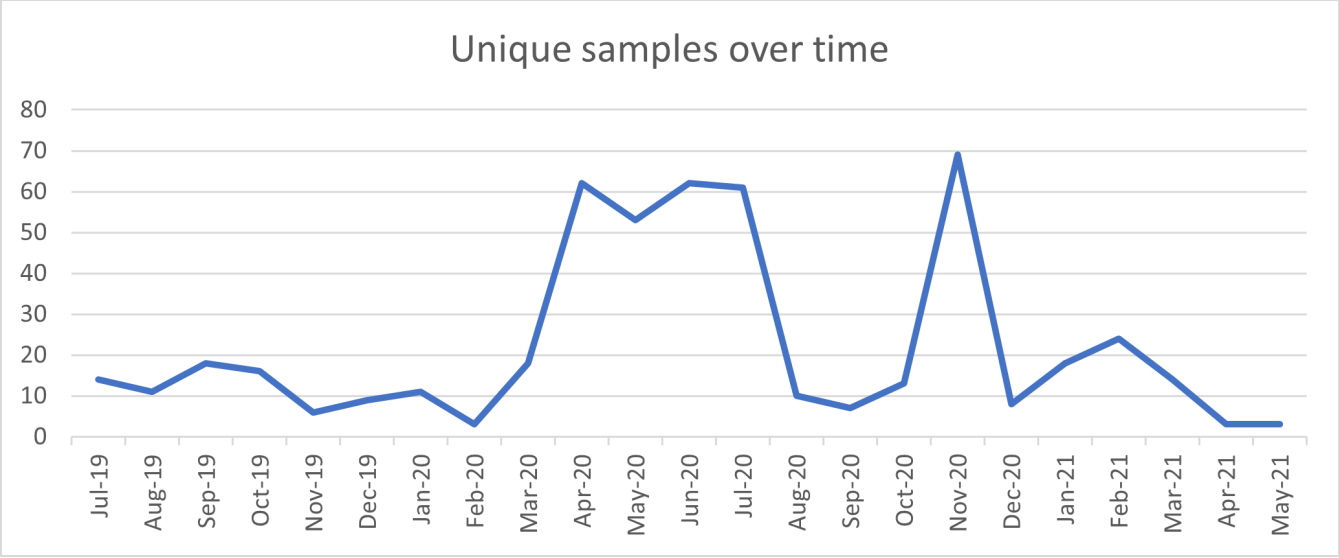
Payload and Configuration Extraction

To automatically extract the payload and configuration of a given loader, one can recreate the decryption mechanism in a language of choice, get the encrypted data from the loader, and decrypt it. The downside here is the need for an exact copy of the decryption mechanism. If the key were to change, or a slightly different algorithm were to be used, the copy would also need to reflect those changes. To avoid dealing with the decryption method, a different approach can be taken.

This loader mistakenly uses static variables to store the decrypted payload and configuration in. In short, these variables are initialized prior to the execution of the main function of the loader. As such, it is possible to reflectively obtain the value of the two variables in question. A detailed how-to guide can be found on my personal website. The data that was extracted from the 513 samples in the set is discussed in the next section.

Bulk Analysis Results

The complete set consists of 513 samples, all of which were found using a single Yara rule. The rule focuses on the embedded resource which is used to persist the loader as a scheduled task on the victim's system. In some cases, the Yara rule will not match a sample, as the embedded resource is obfuscated using ConfuserEx (one example being SHA-256 0427ebb4d26dfc456351aab28040a244c883077145b7b529b93621636663a812). To deobfuscate, one can use ViRb3's de4dot-cex fork of de4dot. The Yara rule will match with the deobfuscated binary. The graph below shows the number of unique samples over time.



The dates are based on VirusTotal's first seen date. Granted, this date does not need to represent the day the malware was first distributed. However, when talking about commodity malware that is distributed in bulk, the date is reliable enough.

The sample set that was used is smaller than the total amount of loaders that have been used in the wild. This loader is often not the first stage, but rather an in-memory stage launched by another loader. Practically, the sample set is sizable enough for this research, but it should be noted that there are more unique loader samples in the wild for the given date range than are used in this report.

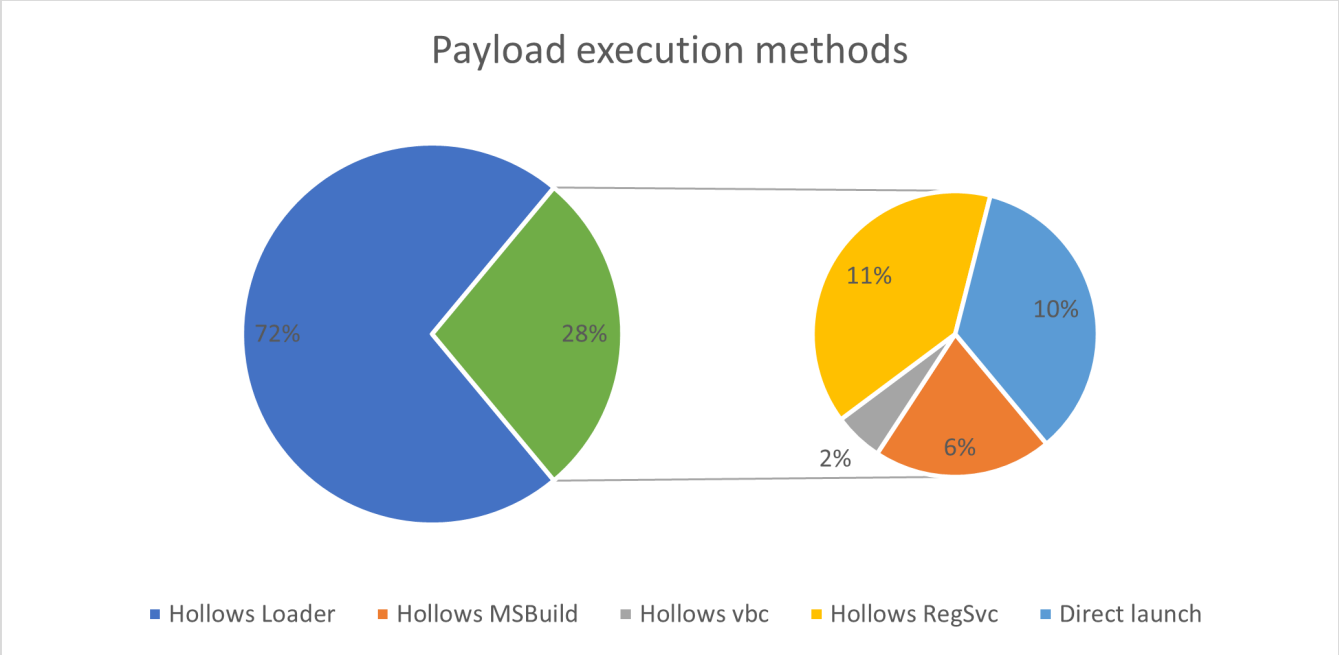
It is useful to know what the capabilities of a single sample are, but the main area of interest of this research is based upon the analysis of all samples in the set. Several features will be discussed, along with thoughts on them. In this section, all percentages refer to the total of 513 unless otherwise specified.

Widespread Usage

The loader's usage is widespread, without a direct correlation towards a specific group or geographical region. Even though some reports mention a specific actor using or creating this loader, the fact that at least one builder has leaked makes attribution to one or more actors difficult. Coupled with the wide variety of targeted industries, as well as the broad geographic targeted areas, it looks like several actors utilise this loader. The goal of this research is not to dig into the actors who utilise this loader, but rather to look at the sample set in general. Appendix A provides a non-exhaustive list of public articles that (at least) mention this loader, in descending chronological order.

Execution Methods

The two options to launch a payload, either reflectively or via process hollowing, are widely apart in usage: 90% of all loaders uses process hollowing, whereas only 10% of the samples are launched via reflection. Older versions of the loader sometimes used to reflectively load a decrypted stager from the loader's resources, which would then launch the loader's payload via process hollowing. The metrics below do not reflect this, meaning the actual percentage of direct launches might be slightly lower than is currently stated. The details can be viewed in the graph below.



Note that the reflective loading mechanism will default to the process hollowing of a new instance of the loader if any exception is thrown. Only DotNet based files can be loaded reflectively, meaning that other files that are executed this way will be loaded using a hollowed instance of the loader.

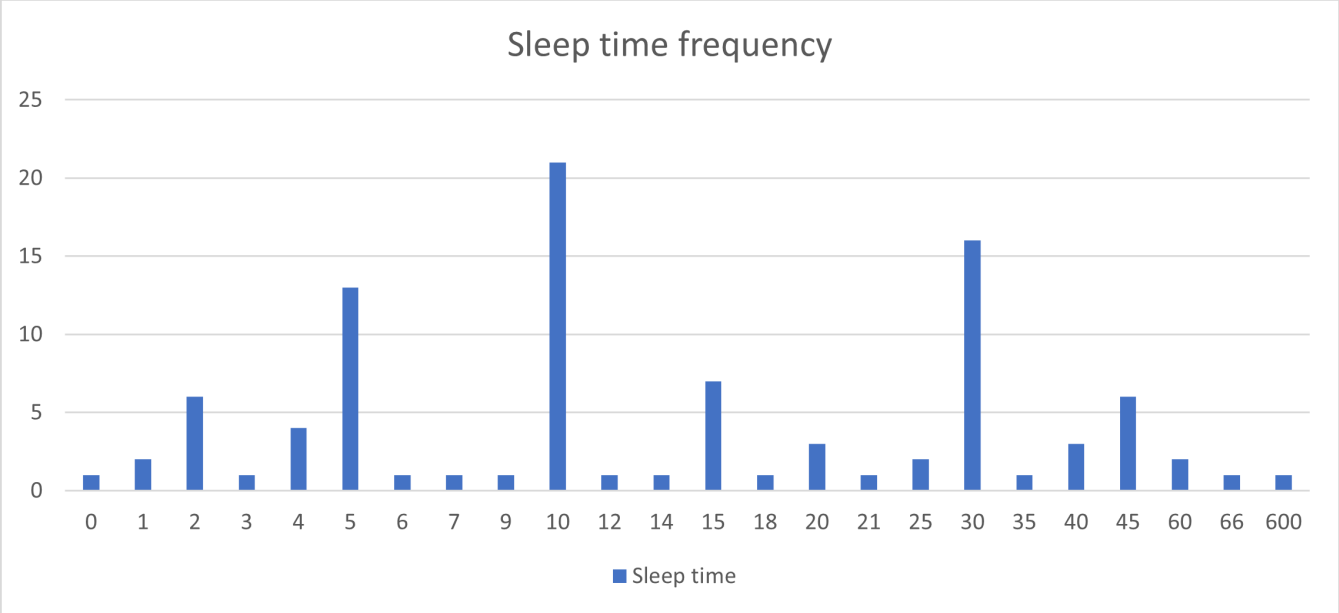
Persistence and Mutexes

The persistence method, which uses a scheduled task to start the loader once the computer boots, is used by 54% of the loaders. This does not mean that the other 46% of the samples are not persisted on the victim's machine, as a different stage could provide persistence as well. Notable is the date within the scheduled task, which equals 2014-10-25T14:27:44.8929027. This date is, at the time of writing, nearly 2500 days ago. If any of the systems in an organization encounter a scheduled task with this exact date, it is wise to verify its origin, as well as the executable that it points to.

A third of all loaders are configured to avoid running when an instance is already active using a mutex. Similar to the persistence mechanism, a mutex could be present in a different stage, though this is not necessarily the case. The observed mutexes seem to consist of only unaccented alphabetical letters, or `[a-zA-Z]+` when written as a regular expression.

Delayed Execution

Delayed execution is used by nearly 37% of the samples, roughly half of which are config_29, meaning this setting was not configurable when creating the sample. The samples where the delayed execution was configurable, equal nearly 19% of the total. On average, a 4 second delay is used. The highest observed delay is 600 seconds. The graph below shows the duration of the delay, and the frequency.



Note that one loader was configured to have a delay of 0 seconds, essentially not delaying the execution. In most cases, the delayed time is a value that can be divided by five, which is often seen as a round number by humans.

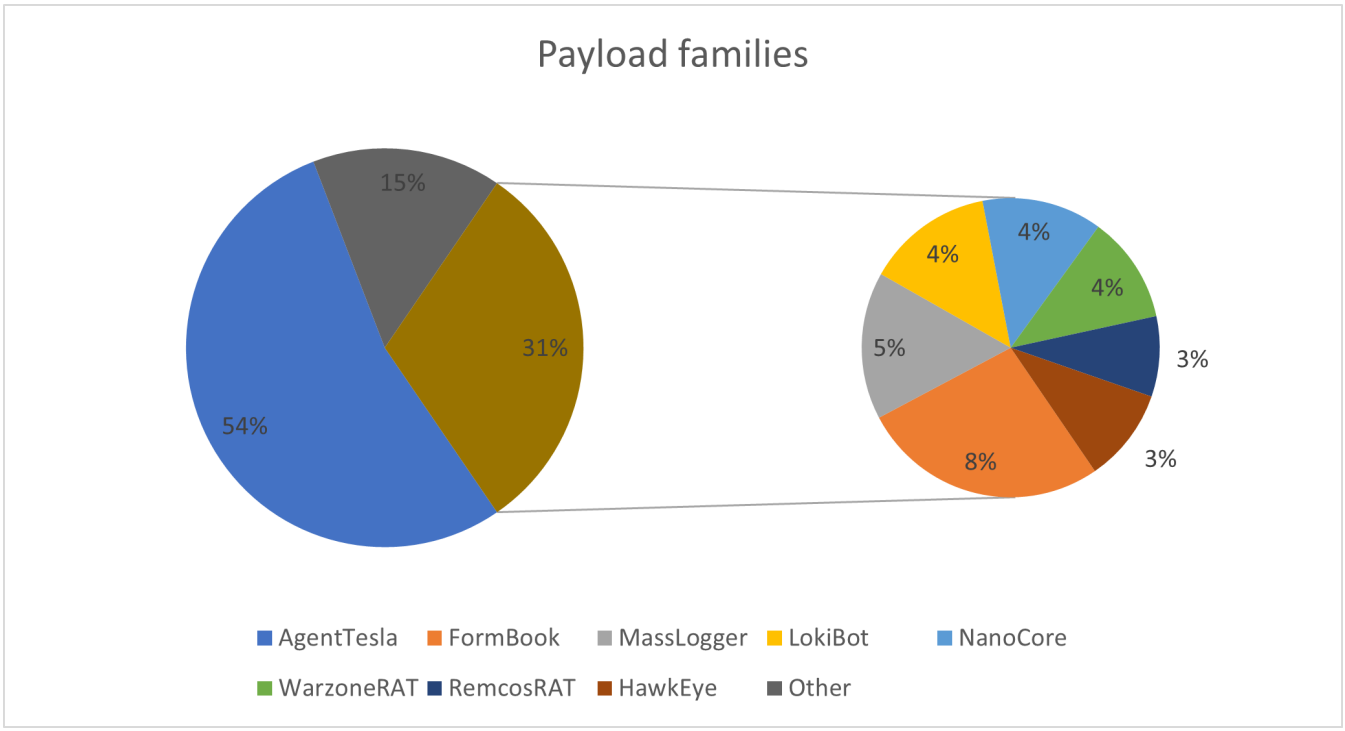
Environmental Awareness

Prior to launching the payload, the loader can perform several checks. A virtual environment can be detected, as well as a sandbox. Roughly 10% of the samples check for the presence of a virtual machine, whereas roughly 11% check if it is executed in a sandbox. Roughly 8% of the 513 samples check for the presence of both, prior to continuing their execution. In other words, 88% of the samples that try to detect a virtual machine, also attempted to detect a sandbox. Vice versa, 74% of the samples that attempted to detect the sandbox, attempted to detect if they were executed on a virtual machine.

The option to disable Windows Defender was mainly present in the earlier samples, which is why only 15% of the set attempts to disable it.

Payload Families

The loader's final goal is to execute the next stage on the victim's machine. Knowing what kind of malware families are often dropped can help to find the biggest pain points in your organization's additional defensive measures. The chart below provides insight into the families that were observed the most. The segment named *other* contains all samples that would otherwise clutter the overview due to the few occurrences per family, such as the RedLine stealer, Azorult, or the lesser known MrFireMan keylogger.



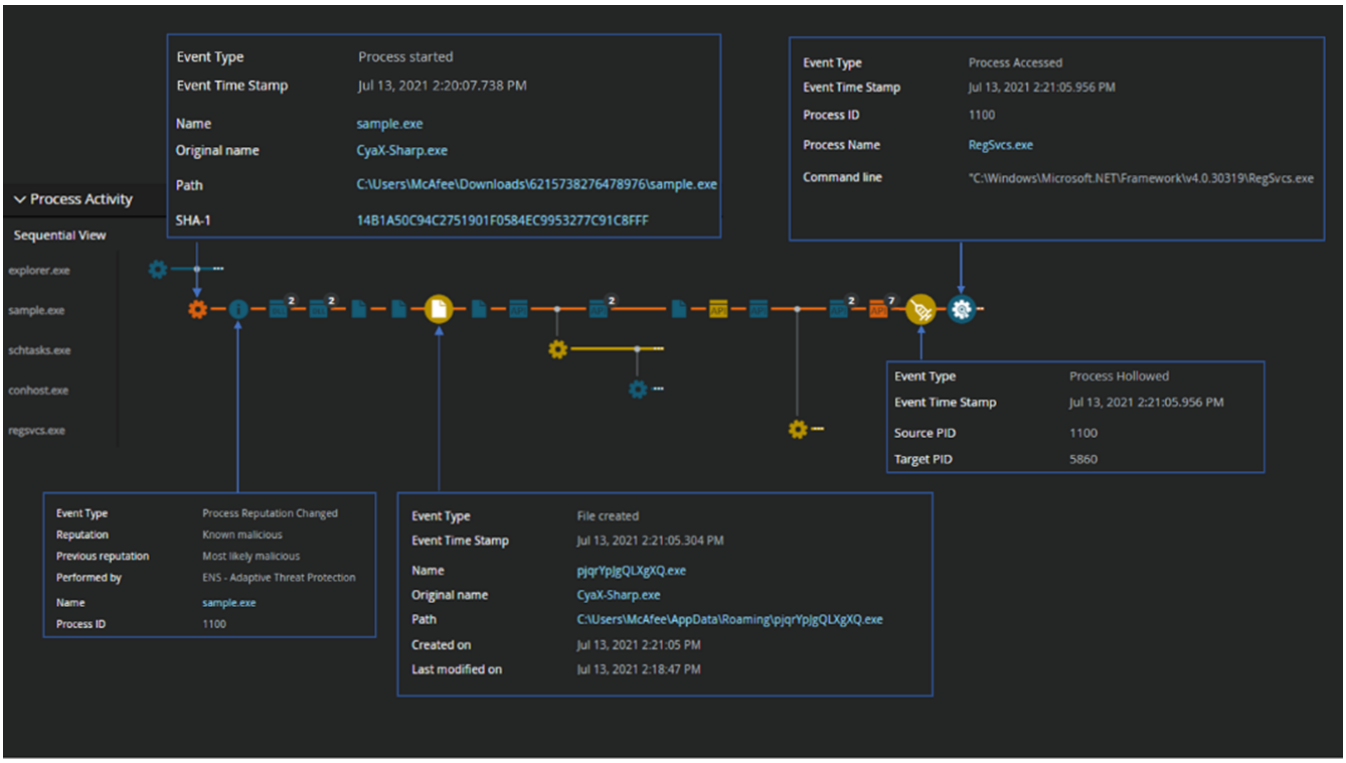
The percentages in the graph are based on 447 total payloads, as 66 payloads were duplicates. In other words, 66 of the unique loaders dropped a non-unique payload. Of all families, AgentTesla is the most notable, both in terms of frequency and in terms of duplicate count. Of the 66 duplicates, 48 were related to AgentTesla.

Barely Utilized Capabilities

Two functions of the loader that are barely used are the message box and the download of a remote payload. The usage of both is, respectively, 1.3% and 0.8%. All of the remote payloads also contained an embedded payload, although one of the four remotely fetching loaders does not contain a URL to download the remote payload from. The external file can be used as an additional module for a next stage, a separate malicious payload, or it can be used to disable certain defense mechanisms on the victim's device.

Conclusion

Companies using the aforementioned onion security model benefit greatly from the dissection of such a loader, as their internal detection rules can be improved with the provided details. This stops the malware's execution in its tracks, as is shown in the sequential diagram of McAfee's detection below.



The techniques that this loader uses are commonly abused, meaning that the detection of a technique such as process hollowing will also prevent the successful execution of numerous other malware families. McAfee's Endpoint Security (ENS) and Endpoint Detection & Response (EDR) detect the CyaX-Sharp loader every step of the way, including the common techniques it uses. As such, customers are protected against a multitude of families based on a program's heuristics.

Appendix A – Mentions of CyaX-Sharp and ReZer0

Below, a non-exhaustive chronologically descending list of relevant articles is given. Some articles contain information on the targeted industries and/or target geographical area.

- On the 12th of January 2021, ESET mentioned the loader in its Operation Spalax blog
- On the 7th of December 2020, ProofPoint wrote about the decryption mechanisms of several known .NET based packers
- On the 5th of November 2020, Morphisec mentioned a packer that looks a lot like this loader
- On the 6th of October 2020, G Data mentioned the packer (or a modified version)
- On the 29th of September 2020, ZScaler mentioned the packer
- On the 17th of September 2020, I wrote about the automatic payload and config extraction of the loader
- On the 16th of September 2020, the Taiwanese CERT mentioned the loader in a digital COVID-19 threat case study
- On the 23rd of July 2020, ClamAV mentioned the loader in a blog
- On the 14th of May 2020, Security firm 360TotalSecurity links the loader to the threat actor Vendetta
- On the 21st of April 2020, Fortinet provided insight into the loader's inner workings
- On the 1st of March 2020, RVSEC0N mentioned the loader
- On the 4th of December 2019, Trend Micro provided a backstory to CyaX-Sharp
- On the 22nd of March 2019, 360TotalSecurity gave insight into some of the loader's features

Appendix B – Hashes

The hashes that are mentioned in this blog are listed below, in order of occurrence. The SHA-1 and SSDeep hashes are also included. A full list of hashes for all 513 samples and their payloads can be found here.

Sample 1

SHA-256: a15be1bd758d3cb61928ced6cdb1b9fa39643d2db272909037d5426748f3e7a4

SHA-1: 14b1a50c94c2751901f0584ec9953277c91c8fff

SSDeep: 12288:sT2BzlxlBrB7d1THL1KEZ0M4p+b6m0yn1MX8Xs1ax+XdjD3ka:O2zBrB7dlHxv0M4p+b50yn6MXsSovUa

Sample 2

SHA-256: 43d334c125968f73b71f3e9f15f96911a94e590c80955e0612a297c4a792ca07

SHA-1: d6dae3588a2a6ff124f693d9e23393c1c6bcef05

SSDeep:
24576:EyOxMKD09DLjhXKCfJIS7fGVZsjUDoX4h/Xh6EkRlVMd3P4eEL8PrZzgo0AqKx/6:EyycPJvTGVijUDlhfEEIUvEL8PrZx0AQ

Sample 3

SHA-256: 0427ebb4d26dfc456351aab28040a244c883077145b7b529b93621636663a812

SHA-1: 8d0bfb0026505e551a1d9e7409d01f42e7c8bf40

SSDeep: 12288:pOIcEfbJ4Fg9ELYTd24xkODnya1QFHWV5zSVPjgXSGHmI:EEj9E/va