Automatic ReZer0 payload and configuration extraction

maxkersten.nl/binary-analysis-course/analysis-scripts/automatic-rezer0-payload-and-configuration-extraction/

This article was published on the 17th of September 2020. This article was updated on the 8th of December 2021.

Understanding how a loader works shortens the time an analyst needs when it is encountered again. It also allows an analyst to create detection rules. This is, however, still manual work. Automating the extraction of the payload and possible configuration of a loader is the ideal scenario. This article covers the automatic extraction of both, based on the ReZer0 loader analysis which was analysed earlier on in this course.

As a follow-up article, I dug deep into this loader's details, as well as historical versions, can be found on McAfee's Advanced Research Team's blog.

Table of contents

The approach

As is described in the analysis, the loader's payload is stored within a private static byte array, whereas the configuration is stored in a private static string array. The required fields are populated based on the string array, which the loader then uses to determine which functions need to be executed.

Within the Dot Net framework, the Assembly class can be used to interact with classes, functions, and fields within a Dot Net binary.

Note that both required fields are *static*, meaning they are assigned their value when the file is loaded. As such, one can iterate over the private static fields within all classes, as the variable and class names are randomised per sample. Once a byte array or string array is found, additional checks can be performed. If these checks are passed, one can store the payload on the disk and print the loader's configuration.

Writing the extractor

Based on prior research, the payload is an executable, meaning one can assume it starts with the *MZ* header. The size of the string array is 37, meaning all other string arrays can be ignored. These checks, especially the second one, may change in the future. If this is the case, or to avoid missing future samples, one could dump all occurrences that are encountered. The code is given in pieces, where each part is explained in the usual step-by-step manner.

Configuring the project

The extractor will be written in C# using Visual Studio 2019. Other versions should work as well, but might require different or additional steps. After making a console application, one has to edit the *App.config* file to allow the project to load code from remote sources. The code snippet to add to the configuration file is given below.

```
<runtime>
    <loadFromRemoteSources enabled="true"/>
</runtime>
```

The reason as to why this is required, is the code that is executed within the ReZer0 loader when it is loaded. This is also the exact functionality on which the extractor is based.

The main function

To easily extract payloads from loaders in bulk, this extractor will go over each file in a given directory. As such, the program requires a single command-line argument: the path to a folder that contains the loaders that are to be analysed. Tests of this program as a whole were done based upon on the 215 samples that were found during prior research. The main function is given below.

```
static void Main(string[] args)
{
        //Get all files in the given folder
        string[] files = Directory.GetFiles(args[0]);
        //Loop through all files in the given folder
        foreach (string file in files)
        {
                //Handle each file
                HandleFile(file);
        }
        //Keep the window open once all files have been iterated through
        Console.ReadKey();
}
```

Extracting the payload

To extract the payload, one must load the loader sample as an assembly in C#. This does not execute the entry point of the binary, which is essential in this case. Do note that technically, all static code is executed by loading the binary. Once the loader sample is loaded, one needs to go over all classes, as the class name that contains the payload is unknown. For each class, one wants to obtain all *private static* fields, since the payload is defined as such. The type of the field is then matched for each encountered field. If the payload contains *MZ* as the first two bytes, it is safe to assume that the payload has been found. Writing it to the disk then preserves the decrypted payload. The payload is already decrypted, as it is a *static* field in the loader, meaning the value is given to the field once the executable is loaded, and prior to the execution of the main function. The code to do so is given below.

```
//Load the ReZer0 loader file
Assembly assembly = Assembly.LoadFile(file);
//Loop over all classes
foreach (Type type in assembly.GetTypes())
{
        //Get all nonpublic static fields from each class
        FieldInfo[] fields = type.GetFields(BindingFlags.NonPublic |
BindingFlags.Static);
        //Loop over all fields
        foreach (FieldInfo fieldInfo in fields)
        {
                //Get the value of each field, as the name is randomised for the
loader
                object value = fieldInfo.GetValue(null);
                //If the type of the current field's value is a byte array, its the
payload, as there is only 1 embedded in the class
                if (value is Byte[])
                {
                        //Create a local variable to more easily handle it
                        byte[] payload = (byte[])value;
                        //Verify that the byte array is a PE file
                        if (payload[0] == 0x4d && payload[1] == 0x5a)
                        {
                                //Write the payload to the disk
                                File.WriteAllBytes(file + "_extracted", payload);
                                Console.WriteLine("Wrote payload to disk as " + file
+ "_extracted");
                                Console.WriteLine("Payload size: " + payload.Length +
" bytes");
                        }
                }
```

Extracting the configuration

To extract the configuration array, one can use the same loop to iterate through all fields. In this case, the requested field type is a string array. If the type matches, and the length is equal to 37, which is the length that is used for the configuration array in the loader, one can assume the configuration array has been found. All that rests then, is to print the configuration value with their corresponding settings. The knowledge based on this is based on the previous research into this loader. The code for the configuration extraction is given below.

```
//If the value type is a string array, it means that the raw configuration values of
the loader
if (value is String[])
{
        //Create a local variable based on the value, casting the type safely due to
the previous if-statement
        String[] settings = (String[])value;
        //If the length of the string array is equal to 37, it is safe to assume that
the configuration array has been found
        //The length is based on prior research
        if (settings.Length == 37)
        {
                //Create and instantiate the output string
                String output = "Payload launch method: ";

                //Get the launch enum value
                int launchEnum = Conversions.ToInteger(settings[0]);
                //If the launch enum value is equal to 4, the payload is launched
directly
                if (launchEnum == 4)
                {
                        output += "launch from loader's memory";
                }
                //In other cases, it executes the payload via a hollowed process
                else
                {
                        output += "process hollowing into ";
                        //Depending on the value, the loader uses a specific process
                        if (launchEnum == 0)
                        {
                                output += "the loader's process";
                        }
                        else if (launchEnum == 1)
                        {
                                output += "MSBuild.exe";
                        }
                        else if (launchEnum == 2)
                        {
                                output += "vbc.exe";
                        }
                        else if (launchEnum == 3)
                        {
                                output += "RegSvcs.exe";
                        }
                }
                //Add a newline for readability
                output += "\n";

                //Get the value of the scheduled task setting
                int shouldSetScheduledTask = Conversions.ToInteger(settings[1]);
                //Print the scheduled task value
                output += "Sets a scheduled task: " + shouldSetScheduledTask + "\n";

                //Get the remote payload execution setting's value
                int shouldExecuteRemotePayload = Conversions.ToInteger(settings[4]);
                //Display the value
                output += "Executes remote payload: " + shouldExecuteRemotePayload +
"\n";
                //if the setting is enabled, the specific settings are read and
printed
                if (shouldExecuteRemotePayload == 1)
                {
                        string url = settings[5];
                        string downloadedFileName = settings[6];
                        output += "URL: " + url + "\n";
                        output += "File name on victim's machine: " +
downloadedFileName + "\n";
                }

                //Gets the anti-virtualisation detection setting
                int shouldDetectVirtualEnvironments =
Conversions.ToInteger(settings[7]);
                //Prints the setting
                output += "Exits when in a virtual environment: " +
shouldDetectVirtualEnvironments + "\n";

                //Gets the anti-sandbox setting
                int shouldDetectSandboxes = Conversions.ToInteger(settings[8]);
                //Prints the setting
                output += "Exits when in a sandboxes: " + shouldDetectSandboxes +
"\n";

                //Gets the messabox display setting
                int shouldDisplayMessageBox = Conversions.ToInteger(settings[29]);
                //Prints the setting's value
                output += "Displays messagebox: " + shouldDisplayMessageBox + "\n";
                //If the setting is enabled, all details are printed
                if (shouldDisplayMessageBox == 1)
                {
                        string messageBoxTitle = settings[30];
                        string messageBoxText = settings[31];
                        int messageBoxButtonsStyle =
Conversions.ToInteger(settings[32]);
                        int messageBoxIconStyle =
Conversions.ToInteger(settings[33]);
                        output += "\tTitle: " + messageBoxTitle + "\n";
                        output += "\tText: " + messageBoxText + "\n";
                        output += "\tButtons style: " + messageBoxButtonsStyle +
"\n";

                        output += "\tIcon style: " + messageBoxIconStyle + "\n";
                }

                //Get the sleep setting
                int sleepTime = Conversions.ToInteger(settings[34]);
                //Print the setting
                output += "Uses sleep to evade detection: " + sleepTime + "\n";
                //If the setting is enabled, the sleep duration is also printed
                if (sleepTime == 1)
                {
                        int sleepDuration = Conversions.ToInteger(settings[35]);
                        output += "Sleeps for " + sleepDuration + " seconds\n";
                }
                //Add a newline for readability
                output += "\n";
                Console.WriteLine(output);
        }
}
```

Running the extractor

Upon putting all the pieces together, one can run the program to iterate all files within a folder. In this test, the files were located at *C:\*. The output of the extractor for several files is given below.

```
Parsing C:\01a083f468e17d5da38d15907e26a71ce4ec6ee575aa5069e090e3d325f855ce
Wrote payload to disk as
C:\01a083f468e17d5da38d15907e26a71ce4ec6ee575aa5069e090e3d325f855ce_extracted
Payload size: 150016 bytes
Payload launch method: process hollowing into MSBuild.exe
Sets a scheduled task: 0
Executes remote payload: 0
Exits when in a virtual environment: 0
Exits when in a sandboxes: 0
Displays messagebox: 0
Uses sleep to evade detection: 0


Parsing C:\03c27b0f45e222123d76ed5a54538bb27ae2739644567985c8f52216b783116d
Wrote payload to disk as
C:\03c27b0f45e222123d76ed5a54538bb27ae2739644567985c8f52216b783116d_extracted
Payload size: 774144 bytes
Payload launch method: process hollowing into the loader's process
Sets a scheduled task: 1
Executes remote payload: 0
Exits when in a virtual environment: 1
Exits when in a sandboxes: 1
Displays messagebox: 0
Uses sleep to evade detection: 0


Parsing C:\03cb9a030d70871b55b2b60be423496a52d536dbd07a94b5597d7395cd0ec130
Wrote payload to disk as
C:\03cb9a030d70871b55b2b60be423496a52d536dbd07a94b5597d7395cd0ec130_extracted
Payload size: 126976 bytes
Payload launch method: process hollowing into vbc.exe
Sets a scheduled task: 1
Executes remote payload: 0
Exits when in a virtual environment: 0
Exits when in a sandboxes: 0
Displays messagebox: 0
Uses sleep to evade detection: 1
Sleeps for 10 seconds
```

Conclusion

Based on the analysis and the usage of reflection within the Dot Net framework, it is possible to extract the required data from a given ReZer0 loader without executing it. Changing the encryption (and therefore the decryption) routine does not make a difference in this scenario, as the fields are already assigned a value once the file is loaded, effectively bypassing the complete encryption module of the loader.

Extractor code

The complete code for the extractor is given below. Keep the aforementioned *App.config* changes in mind, and import the Visual Basic reference correctly in order to make the code work.

```
using System;
using System.IO;
using System.Reflection;
using Microsoft.VisualBasic.CompilerServices;

namespace ReZer0_extractor
{
    /// <summary>
    /// ReZer0 payload and settings extractor by Max 'Libra' Kersten (@Libranalysis
on Twitter, https://maxkersten.nl)
    /// Licensed under GPLv3 (https://www.gnu.org/licenses/gpl-3.0.en.html)
    /// </summary>
    class Program
    {
        /// <summary>
        /// Extracts the payload from the loader to a file with the same name with
"_extracted" appended to it.
        /// The config of the loader is printed to the console.
        /// </summary>
        /// <param name="file"></param>
        static void HandleFile(string file)
        {
            //Print the file path to indicate which file is being processed
            Console.WriteLine(@"Parsing " + file);

            //Load the ReZer0 loader file
            Assembly assembly = Assembly.LoadFile(file);
            //Loop over all classes
            foreach (Type type in assembly.GetTypes())
            {
                //Get all nonpublic static fields from each class
                FieldInfo[] fields = type.GetFields(BindingFlags.NonPublic |
BindingFlags.Static);
                //Loop over all fields
                foreach (FieldInfo fieldInfo in fields)
                {
                    //Get the value of each field, as the name is randomised for the
loader
                    object value = fieldInfo.GetValue(null);
                    //If the type of the current field's value is a byte array, its
the payload, as there is only 1 embedded in the class
                    if (value is Byte[])
                    {
                        //Create a local variable to more easily handle it
                        byte[] payload = (byte[])value;
                        //Verify that the byte array is a PE file
                        if (payload[0] == 0x4d && payload[1] == 0x5a)
                        {
                            //Write the payload to the disk
                            File.WriteAllBytes(file + "_extracted", payload);
                            Console.WriteLine("Wrote payload to disk as " + file +
"_extracted");
                            Console.WriteLine("Payload size: " + payload.Length + "
bytes");
                        }
                    }
                    //If the value type is a string array, it means that the raw
configuration values of the loader
                    if (value is String[])
                    {
                        //Create a local variable based on the value, casting the
type safely due to the previous if-statement
                        String[] settings = (String[])value;
                        //If the length of the string array is equal to 37, it is
safe to assume that the configuration array has been found
                        //The length is based on prior research
                        if (settings.Length == 37)
                        {
                            //Create and instantiate the output string
                            String output = "Payload launch method: ";

                            //Get the launch enum value
                            int launchEnum = Conversions.ToInteger(settings[0]);
                            //If the launch enum value is equal to 4, the payload is
launched directly
                            if (launchEnum == 4)
                            {
                                output += "launch from loader's memory";
                            }
                            //In other cases, it executes the payload via a hollowed
process
                            else
                            {
                                output += "process hollowing into ";
                                //Depending on the value, the loader uses a specific
process
                                if (launchEnum == 0)
                                {
                                    output += "the loader's process";
                                }
                                else if (launchEnum == 1)
                                {
                                    output += "MSBuild.exe";
                                }
                                else if (launchEnum == 2)
                                {
                                    output += "vbc.exe";
                                }
                                else if (launchEnum == 3)
                                {
                                    output += "RegSvcs.exe";
                                }
                            }
                            //Add a newline for readability
                            output += "\n";

                            //Get the value of the scheduled task setting
                            int shouldSetScheduledTask =
Conversions.ToInteger(settings[1]);
                            //Print the scheduled task value
                            output += "Sets a scheduled task: " +
shouldSetScheduledTask + "\n";

                            //Get the remote payload execution setting's value
                            int shouldExecuteRemotePayload =
Conversions.ToInteger(settings[4]);
                            //Display the value
                            output += "Executes remote payload: " +
shouldExecuteRemotePayload + "\n";
                            //if the setting is enabled, the specific settings are
read and printed
                            if (shouldExecuteRemotePayload == 1)
                            {
                                string url = settings[5];
                                string downloadedFileName = settings[6];
                                output += "URL: " + url + "\n";
                                output += "File name on victim's machine: " +
downloadedFileName + "\n";
                            }

                            //Gets the anti-virtualisation detection setting
                            int shouldDetectVirtualEnvironments =
Conversions.ToInteger(settings[7]);
                            //Prints the setting
                            output += "Exits when in a virtual environment: " +
shouldDetectVirtualEnvironments + "\n";

                            //Gets the anti-sandbox setting
                            int shouldDetectSandboxes =
Conversions.ToInteger(settings[8]);
                            //Prints the setting
                            output += "Exits when in a sandboxes: " +
shouldDetectSandboxes + "\n";

                            //Gets the messabox display setting
                            int shouldDisplayMessageBox =
Conversions.ToInteger(settings[29]);
                            //Prints the setting's value
                            output += "Displays messagebox: " +
shouldDisplayMessageBox + "\n";
                            //If the setting is enabled, all details are printed
                            if (shouldDisplayMessageBox == 1)
                            {
                                string messageBoxTitle = settings[30];
                                string messageBoxText = settings[31];
                                int messageBoxButtonsStyle =
Conversions.ToInteger(settings[32]);
                                int messageBoxIconStyle =
Conversions.ToInteger(settings[33]);
                                output += "\tTitle: " + messageBoxTitle + "\n";
                                output += "\tText: " + messageBoxText + "\n";
                                output += "\tButtons style: " +
messageBoxButtonsStyle + "\n";
                                output += "\tIcon style: " + messageBoxIconStyle +
"\n";
```

```
                            }

                            //Get the sleep setting
                            int sleepTime = Conversions.ToInteger(settings[34]);
                            //Print the setting
                            output += "Uses sleep to evade detection: " + sleepTime +
"\n";

                            //If the setting is enabled, the sleep duration is also
printed
                            if (sleepTime == 1)
                            {
                                int sleepDuration =
Conversions.ToInteger(settings[35]);
                                output += "Sleeps for " + sleepDuration + "
seconds\n";
                            }
                            //Add a newline for readability
                            output += "\n";
                            Console.WriteLine(output);
                        }
                    }
                }
            }
        }

        /// <summary>
        /// ReZer0 payload and settings extractor by Max 'Libra' Kersten
(@Libranalysis on Twitter, https://maxkersten.nl)
        /// Licensed under GPLv3 (https://www.gnu.org/licenses/gpl-3.0.en.html)
        /// </summary>
        static void Main(string[] args)
        {
            //Get all files in the given folder
            string[] files = Directory.GetFiles(args[0]);
            //Loop through all files in the given folder
            foreach (string file in files)
            {
                //Handle each file
                HandleFile(file);
            }
            //Keep the window open once all files have been iterated through
            Console.ReadKey();
        }
    }
}
```